BBC MASTER GUIDES

MASTERING INTERPRETERS AND COMPILERS

Techniques for the BBC Model B, Master & Electron

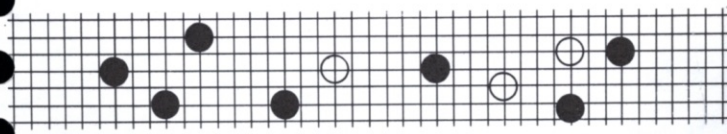
BRUCE SMITH

BBC MASTER GUIDES

MASTERING INTERPRETERS AND COMPILERS

BRUCE SMITH

BBC SOFT



Throughout this book the letters 'BBC' refer to the British Broadcasting Corporation. The term 'BBC Micro' refers to the micro-computer manufactured by Acorn Computer plc under licence from the BBC. 'Tube' and 'Econet' are registered trade marks of Acorn Computer plc.
All rights reserved. No part of this book (except for brief passages quoted for critical purposes) may be reproduced or translated in any form or by any means, electronic, mechanical or otherwise, without the prior written consent of the copyright owner.
Disclaimer: Because neither the BBC nor the author have any control over the circumstances of use of the book, no warranty is given or implied as to its suitability for any particular application. No liability can be accepted for any consequential loss or damage, however caused, arising as a result of the information and advice given in this book.

Acknowledgements

No book is the work of a single person, and in this case I am in debt to many of my good friends: to everyone at BBC Publications for all their hard work and effort, in particular Meyer Solomon's editing skills; to Kitty Milne for her work and comments on the original manuscript; to Dave Acton for his sterling work on the TACTLESS Compiler and subsequent comments – including marks out of ten! Finally, but certainly not least, for the patience of my wife with me during the final months of her pregnancy and the birth of our son, Marc, to whom this book is dedicated with immense pride.

© Bruce Smith 1986
Edited by Meyer N. Solomon
Technical assistance by David Atherton
First published 1987

Published by BBC Soft,
a division of BBC Enterprises Limited,
80 Wood Lane, London W12 0TT
Typeset in 10/12 Rockwell by
Rowland Phototypesetting Limited,
Bury St Edmunds, Suffolk
and printed in Great Britain by
St Edmundsbury Press Limited,
Bury St Edmunds, Suffolk.

Covers printed by Malvern Press Limited, London.
Bound by James Burn Oxford, Eynsham, Oxford.

CONTENTS

Preface 6

INTERPRETERS 9

1 Introduction 10
Comment allez vous? 10

2 The Wedge 12
The CLIV 13

3 The Interpreter 19
The command table 19
An interpretive call 25
Inserting the Wedge 28
A token gesture 29

4 Grafrite – the Language 41
Tools 42
Planning 43
The memory map 45
Vectors and storage 47
Variables 48
Command syntax 49
The program, the errors 52
Writing Grafrite 52
Initialisation 54
Prompting input 56
Line by line 59
Immediate action 60

5 Language? – What Language? 81
Handling BRK 81
Handling BREAK 85

6 Improvements 93
Program editing 95
Escape route 96

Adding new commands 97
Tokenising 99
Repeating a loop 101

THREADED INTERPRETIVE LANGUAGES 105

7 About TILS 106
Searching the dictionary 110
Entering Mini TIL 114
Going THORF 119
Extending Mini THORF 120

8 First, Second, Third 135
Stacks 136
Dictionary format 138
The two interpreters 138
Building dictionary entries 143
Variables and constants 144
Memory configuration 146

9 Polished Polish! 151
RPN 152
The RPN stack 153

COMPILERS 162

10 Introduction 163
A parsing phase 164
The two L's 166

11 Tactless 168
Writing the compiler 168
The Compiler Source Language 170
The data stack 171
Library and variable space 173
The Compiler 173
Macros 176

12 **The Library** 178
The list 178
The LIBINFO file 179
Direct and indirect 181
How to enter the library 183
The library routines 184
Error messages 186

13 **The Editor** 200
Interfacing the library 200
Line to memory 202
The tokeniser 206
The tokenising process 208
Detokenising 211

14 **The number game** 213
Handling numbers 213
Variable variables 218
In the know 221

15 **The TACTLESS language** 224
Direct commands 225
Commands 229
Operators 243

16 **The Compiler** 244
Using TACTLESS 252
Tape users 255
ROM Filing System 256
Anatomy 256
Extending TACTLESS 262

Appendix A 291

Appendix B 297

Appendix C 305

Appendix D 307

Appendix E 309

Index 310

PREFACE

How many times have you phoned the electricity or gas board to query your bill just to be told – 'oh, it's the computer's fault'. Of course it wasn't the computer's fault. It was the fault of the person who wrote the original program. The point is that a computer cannot do anything it hasn't been told to, and it's only capable of interpreting what it has to do by the commands given to it. On most computers, and indeed our beloved BBC micro, this is normally done in BASIC. We write a BBC BASIC program and run it to perform the tasks we require. However, how does BBC BASIC work? The truth is that BBC BASIC is itself a program being run by a very large machine-code program housed in a ROM chip. So what we have is a program running within a program!

This book deals with the understanding and writing of these programs to run programs – Languages, if you prefer. However, it does not trudge through with page after page of theory – first and foremost this book is practical, and theory is demonstrated with programs for you to type in and see for yourself. The first section of this book culminates in the presentation of a graphics language, aptly called Grafrite, which will allow you to write programs using commands which are solely there to generate graphics.

In the second half of the book we deal with the programmer's dream – the Compiler – a program which will generate machine code from your BASIC-type programs, save the machine code and use it at a later stage. Unlike many other Compilers published for the BBC Micro the compiler presented in these pages is totally independent of the BASIC chip. In other words, it will produce stand-alone machine code, and does not make use of routines within the various ROM chips, other than the legal Operating System OS calls.

Over the years since the introduction of the BBC Model A micro there have been new machines and different

versions of BASIC released. This book applies to all these computers and with the exception of the 16k model A all programs will work on the following combinations:

Firmware:
BASIC 1
BASIC 2
BASIC 4
BASIC 5
OS 1.00
OS 1.20
OS 3.00
OS 3.20
OS 5.00
DFS
ADFS
NFS
ANFS

Micros:
BBC B
BBC B+
BBC B+128
BBC with 6502 second processor
Master 128
Master ET
Master Turbo
Master Compact
Electron
Electron Plus 1
Electron Plus 1/3

By the nature of the programs themselves, the listings presented are in many cases very long – in fact, the Compiler could comprise the longest listings you have typed in! Where possible, therefore, checksums are given to help validate that your program is correct. The programs are all listed direct from working and independently tested copies on disc, so 99% reliability is assured. If the length of the listings is daunting, a disc containing all the programs from this tome plus a few extras can be bought – full details can be found in Appendix E.

The programs, though written with a disc-based machine, will work with the cassette filing system, but attention should be paid to the notes aimed at tape users, when these are found in the text. Users of the Electron Plus 3 will find severe memory restrictions when using the Compiler and again attention should be paid to the notes found in those later chapters.

All programs have been written in BASIC 1 and are thus compatible with all succeeding versions of that language. However, users of later BASICs could replace the EQU simulations with the resident EQU commands if so desired. Thus assembler references to:

```
OPT FNequb
```

could be replaced with a simple

```
EQUB
```

and the corresponding FN deleted from the program.

Important note: The listings of the main programs in this book are to be found at the end of each chapter.

I would be very interested to see the results of your own programs developed with the routines in this book, and any improvements you make. Send them to me care of the address given below, on a disc with return postage. I would then hope to return your disc with copies of other people's efforts at a later date so that we can all benefit.

Bruce Smith
December 1986
PO Box 19 London
N11 1DS

INTERPRETERS

CHAPTER ONE

INTRODUCTION

Comment allez vous?

That's about as far as my rudimentary French will let me go. (If you know three words fewer in French than I do, it means 'Hello. How are you?'). When I need to know any more I refer to my English-French pocket dictionary. Transforming one language into another affords many multi-lingual people their livelihood and the job is that of a translator or interpreter!

An interpreter for a computer does exactly the same job, though on a slightly different level. It takes input from the keyboard or a program which is written in a computer language, converts it into a form that the computer can understand and then acts upon it.

We have introduced another topic here as well – the computer language. A computer language, such as BASIC, is written in a way that it readily understood by the user; however, languages such as BASIC are written in the computer's native language, ie, machine-code. For the computer to understand higher level languages such as BASIC it must be able to interpret the commands specified and translate them into machine code.

All BBC and Electron microcomputers come supplied with a language, BBC BASIC. BASIC allows you to enter programs using a predefined set of words or commands described in the User Guide. These commands follow certain rules which have to be adhered to for them to function correctly. When the program is run the BASIC Interpreter comes into action. It looks at each command in turn and tries to identify it. When it has been identified the interpretation process is complete – all the interpreter need do now is to carry out the task indicated by the command.

Any commands, or strings of commands that are entered which do not adhere to the rules or any unrecognised commands will result in an error being generated by the BASIC Interpreter.

BBC BASIC is written in 6502 machine-code, the native language of the BBC Micro and Electron. As many of you know, it's certainly possible to learn how to program in machine-code. But programming in machine-code means dealing with sequences of numbers that are difficult to learn and read. BBC BASIC includes a machine-code assembler which generates machine-code instructions from a program written in assembly language. Assembly language is itself written in an easily remembered and read mnemonic form. This too has to be interpreted by BASIC. So, as you see, we as programmers rely greatly on languages and their interpreters.

Microcomputers can be multi-lingual also, but unlike humans they don't have to go through any learning process. All that need be done to enable a BBC Micro or Electron to understand a new language is to load the language into the computer's memory and set it working. This normally means plugging in a ROM chip or a cartridge. Alternatively, a language and associated interpreter can be loaded into the main memory from disc or tape. Examples of other computer languages include FORTH, Pascal, BCPL and LISP.

All languages for the BBC and Electron micros have this in common: they are written in machine code, but their rules and commands for programming are totally different.

Over the next few chapters we will be examining how an interpreter works by using practical programming examples and we'll also write a minimal graphics language which you will be able to add to, and enhance, to suit your own needs. But first let's look at how we can set up a very simple interpreter using some of the hooks into the micro's operating system that are provided to do just that.

CHAPTER TWO

THE WEDGE

A mechanism within the BBC micro and Electron operating systems allow their existing operations to be intercepted. There are a number of reasons why you may wish to make this interception. It may be because you wish to change the way a certain part of the micro's operation works; you might want to remove an operation totally; or more likely, you might want to improve the operation.

The way to do this is to change the 'operating system vectors' associated with the particular operation. An operating system vector is an address held within RAM (normally in page two of the memory map) which itself contains an address that is, in effect, the entry point pointer to the routine responsible for interpreting the action desired. Thus, a vector is effectively two locations in memory in which an address is held. This address can then be jumped-to using the machine-code instruction:

JMP (vector)

As an example, imagine that the two bytes at &70, &71 (low, then high) held the address &2200. Performing:

JMP (&70)

would not cause a jump to &70 but instead a jump to the address held in the vector &70, &71 – &2200 in this case. Location &71 holds &22; location &70 holds &00. The advantage of using this indirect approach is that we can access, and jump to, memory locations that may either change while a program is running or not be known at the time of writing the program.

Table 2.1 lists the main OS vectors and their associated addresses.

Vector Address	Vector Name	Action
&200,1	USERV	The User Vector
&202,3	BRKV	The Break Vector
&204,5	IRQ1V	The primary Interrupt Vector
&206,7	IRQ2V	The secondary Interrupt Vector
&208,9	CLIV	The Command Line Interpreter vector
&20A,B	BYTEV	The OSBYTE Vector
&20C,D	WORDV	The OSWORD Vector
&20E,F	WRCHV	The Write Character Vector
&210,1	RDCHV	The Read Character Vector
&212,3	FILEV	Read/Write a File Vector
&214,5	ARGSV	Read/Write Arguments vector
&216,7	BGETV	Read Byte Vector
&218,9	BPUTV	Write Byte Vector
&21A,B	GBPBV	Get/Put block-of-bytes Vector
&21C,D	FINDV	Open/Close file Vector
&21E,F	FSCV	Various file-actions Vector
&220,1	EVENTV	Event Handling Vector
&222,3	UPTV	User Print Vector
&224,5	NETV	Econet Vector
&226,7	VDUV	VDU Vector
&228,9	KEYV	Keyboard Vector
&22A,B	INSV	Insert-into-buffer Vector
&22C,D	REMV	Remove-from-buffer Vector
&22E,F	CNPV	Count/Purge Vector
&230,1	IND1V	Spare
&232,3	IND2V	Spare
&234,5	IND3V	Spare

Table 2.1: The OS Vectors

The CLIV

The vector at &208 is of great interest to us. It is the Command Line Interpreter Vector, CLIV for short. Whenever BASIC (or any language) encounters a star command, ie, one prefixed with a '*', control is passed to the Machine Operating System (MOS) via this vector. Figure 2.1 lists the details of CLIV.

Call address	:	&FFF7
Vectored address	:	&208
Entry Conditions:		
X register	:	low-byte line address
Y register	:	high-byte line address
Exit Conditions:		
Registers	:	all undefined

Figure 2.1: CLI details

The call address, &FFF7, is the entry point of the routine. The vectored address, &208, contains the entry point of CLIV in the two locations &208, &209. *Entry* conditions are what the CLIV routine expects to find in the registers, which is an address in this case. *Exit* conditions are what the registers (A, X, Y and P) contain when the routine has finished its job and returns back to your program.

BASIC makes the OSCLI call by first setting the two index registers (X and Y) to point to the * at the start, in memory, of the star command. It then jumps to &FFF7 (OSCLI), which in turn jumps indirectly to the address held in the vector at &208.

By altering the contents of CLIV – for instance, to point to machine code of your own held in RAM – we can get it to add extra commands or facilities to the system we are using – this is commonly called a *wedge*.

There are two purposes for which a wedge can be written: first, to add extra facilities but not replacing the existing ones; second, to completely replace the existing facilities.

For partial replacement we need to first preserve the original contents of the vector somewhere in memory before re-vectoring, so that the original command can end with an indirect jump to the 'old' address. The contents of all registers should also be saved so they can be restored prior to jumping to the default vector address.

A total replacement should be undertaken carefully and should not generally be via CLIV. However, in such cases you should end your wedge routine with a simple RTS instruction (or RTI if re-vectoring BRKV, IRQ1V or IRQ2V), all of which are entered when the relevant interrupt occurs.

Program 2.1 provides a simple demonstration of re-vectoring CLIV but not totally replacing it. It does nothing more than beep the internal speaker each time a * command is entered. Let's examine the program in some detail. The first two assembly language instructions are PHP and SEI (lines 110 and 120). PHP is required to save the flags in the processor so as to allow them to be restored once the wedge has finished with the command being processed.

The SEI instruction prevents any interrupt requests occurring while the vector's contents are being altered. An interrupt is a signal issued by either the Machine Operating System (MOS), another ROM, another chip, or even an external source; it tells the micro to drop everything it is doing and do something else! Obviously there is some special machine-code to handle this within the MOS itself. Once the interrupt has been 'serviced', control can be handed back to the routine that was interrupted; this can then continue with its task. But while a vector's contents are being changed, the SEI instruction stops this interrupt from occurring, which is important. If an interrupt were allowed to butt in when only the first byte of the vector had been changed, a jump by the interrupt routine through the vector would certainly cause the machine to hang-up and, perhaps, even corrupt vital information.

The current contents of CLIV need to be extracted and saved in a safe place, in this case within the body of the main program. This is fine – the only pre-requisite is that the order of the bytes is maintained (ie low-byte first) so that the new home of these bytes can be used as a vector itself. This is done in lines 130 to 210. Note that in line 140 we are testing to see if the high-byte of CLI is positive – less than &80, if you prefer. This is in effect testing to see if the wedge is already present. Normally the contents of &209, &208 will be pointing into the MOS and as such their high-byte contents will be greater than &C0. When the wedge is run the byte here will be less than &80 as the wedge will be residing in some free RAM – at &900, to be exact. If this test is not made and the wedge is already present, the old CLI value saved will be overwritten with the address of the wedge itself, so that the wedge will simply call and

re-call itself forever!

The final act is to restore the status register contents and re-enable interrupts, with PLP and CLI respectively (lines 220 and 230).

The CLIV is now pointing at *wedge* (line 370). Before and after you have run the program enter,

```
PRINT ~!&208 AND &FFFF
```

to see how the contents of &208,&209 change.

Now each time a * command is performed, control will first pass to *wedge*. Once here, the three registers will be saved into the spare bytes marked by the labels *accum*, *xreg*, and *yreg* – then restored after a beep has been made on the internal speaker, before finally departing back to the default (usual) address of CLIV with an indirect jump to *clilow*. (&FFE3 is the MOS address of the OSASCI routine).

Try it for yourself. First run the program and then type

```
CALL &900
```

Any * command such as *HELP, *CAT, or just a * will first give a beep before going about its assigned task! Now try re-running the assembly program and performing CALL &900.

The error message 'Wedge already present!' will be displayed by using the MOS's own error handling facility. Whenever the MOS encounters a BRK command (opcode &00) it will try to print out an ASCII character string (here it's *wedge*) until it encounters another BRK opcode at line 350. Note that the first byte after the first BRK (line 320) is assumed to be the error code and is the reason why the BRK in line 330 does not inhibit the error printing process – the MOS assumes an error code number 0!

To see what can happen if this sort of error trapping is not performed, hit the BREAK key, type OLD and delete line 140, RUN and CALL &900 twice, and now try entering a * command!

One final point about re-vectoring the OS vectors for those of you using second processors. Changing vectors in the second processor only affects calls which have their origin in the second processor in question. However, changing vectors in the I/O processor (ie the

BBC micro or Electron) will affect calls to the re-vectored routine regardless of whether it originated in the second processor or I/O processor.

Now that we have learned to interrupt the processing of the MOS and add our own wedge to perform a task, albeit a very simple one, we can move on to writing programs to identify our own commands – a subject that's open to interpretation!

Listing 2.1

```
 1  REM PROG2.1 - 14/3/86
 10 REM Re-vectoring OSCLI
 20 REM Mastering Interpreters
 30 REM and Compilers
 40 REM (C) Bruce Smith 1986
 50 :
 60 wedge$="Wedge already present!"
 70 FOR pass=0 TO 3 STEP 3
 80 P%=&900
 90 [OPT pass
100 .revector
110 PHP
120 SEI
130 LDA &209
140 BPL error
150 STA clihigh
160 LDA &208
170 STA clilow
180 LDA #wedge DIV 256
190 STA &209
200 LDA #wedge MOD 256
210 STA &208
220 PLP
230 CLI
240 RTS
250 \
260 .clilow
270 BRK
280 .clihigh
290 BRK
300 \
310 .error
320 BRK
```

```
33Ø  BRK
34Ø  OPT FNequs(wedge$)
35Ø  BRK
36Ø  \
37Ø  .wedge
38Ø  STA accum
39Ø  STX xreg
4ØØ  STY yreg
41Ø  LDA #7
42Ø  JSR &FFE3
43Ø  LDA accum
44Ø  LDX xreg
45Ø  LDY yreg
46Ø  JMP (clilow)
47Ø  \
48Ø  .accum
49Ø  BRK
5ØØ  .xreg
51Ø  BRK
52Ø  .yreg
53Ø  BRK
54Ø  ]
55Ø  NEXT pass
56Ø  END
57Ø  :
58Ø  DEF FNequs(A$)
59Ø  $P%=A$
6ØØ  P%=P%+LEN(A$)
61Ø  =pass
```

CHAPTER THREE

THE INTERPRETER

The command table

For a computer to be able to identify a command or series of commands it must have something to compare them against. All languages have a basic set of commands which will be recognised by the language interpreter – these commands are often called *keywords*. For a language to identify each of these keywords, a table of all them within its own memory area needs to be kept – this table is the *keyword* or *command* table.

In simple terms, when a command is issued a routine within the computer's current language first compares it with each of the commands in its command table. If it fails to recognise the command then the language generates an error message. On the other hand, if the command is identified then control must next be passed to the routine within the language that will perform the task specified by the command. The first stage of the interpretation process is to compare one string – the command string – with a series of strings in memory – the command table. The assembly language coding for this is straightforward. Assuming that *table* marks the start of the command table and *program* is a vector address pointing to the command:

```
LDX  #&FF          \ initialise index into table
LDY  #&FF          \ seed indirect index
.check
INX                \ increment both indexes
INY
LDA (program),Y    \ get byte from command
CMP table,X        \ compare against table
BEQ check          \ branch back for next byte
```

In the above segment, *program* is a vector to the string of commands that is under interpretation. The segment

will continue to repeat while each pair of bytes it tests are identical – let us suppose they are, for the time being. Ultimately the last byte of the command string is reached. For simplicity's sake we'll assume that this is a return character, ASCII 13 (&0D). We could check for this by appending this next segment of code to that above

```
CMP #&0D          \ is a return?
BEQ found         \ yes - found command
BNE error         \ no, it's unrecognised; so
                    error!
```

Even though it seems so simple, the check routine would allow you to identify and call the execution code of a single command!

But to increase these capabilities and to allow our interpreter to identify and execute more than one command, we need to be able to perform the following:

- identify a command
- otherwise, move on to the next command
- recognise when the command is not known
- jump to the identified command's address

We already have the first segment of code (above) to perform the 'identify a command' process. To perform the second, that is, to move on to the next command, we must be able to keep track of two pointers.

The first pointer must be able to access each character in a given command string. Because the address of the command may alter, it's wise to access it by using a vector; so, in this case, the vector points to the start of the command and the Y register is an indirect index from the start as illustrated in figure 3.1.

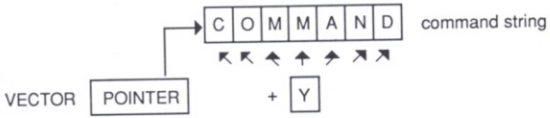


Figure 3.1

The second pointer must allow us to step through the locations of the command table. The command table will normally start at a fixed address and so direct addressing can be used to step through it. However, if the first command in the table does not match the given command then the start address of the second command (and then the third, etc) must be used to try successive comparisons against the given command. To this end, the X register is used as an index to step through the command table.

All that's required to move on to the next command is to move the X register on so that it points to the start of the next command. To do this, we need a method of marking where this one command ends and another begins in the command table. As the command table will be composed of ASCII characters which have a value less than &80, we can use any value equal to or greater than this as a marker – &FF for example. Note that &FF is negative as far as the comparison operation is concerned. So, to move on to the next command we simply employ:

```
.moveon
INX                 \ increment X
LDA table,X         \ get next byte from command
                      table
BPL moveon          \ branch until negative
LDY #&FF            \ negative, so end of command:
                      reinitialise Y to &FF
BMI check           \ and jump back to 'check'
```

The BPL instruction will continue to cause a branch back to *moveon* each time it encounters an ASCII character. The &FF separator, meaning that the byte is not an ASCII character, will set the negative flag of the status register and therefore the branch will fail. Before control can be passed back to the identifying routine at *check* the Y register must be re-initialised to &FF (this is a coincidence, and has nothing to do with the bytes in the command table). This is very important – failure to do this will mean that the vectored address will not be pointing to the start of the command to be identified on re-entry to the interpreter!

A command is defined as not having been recog-

nised by the interpreter when the search for it reaches the very end of the command table. This position can be marked in a variety of ways. For example, by using a zero byte, ie, &00, or even another &FF byte, all we need do is test for it before returning to the *check* routine. Reworking the segment of code above we could adapt it thus:

```
.moveon
INX
LDA table,X
BPL moveon
INX                 \ move X on one byte
LDA table,X         \ get next byte
CMP #&0             \ is it end of command table?
BEQ error           \ yes, unrecognised!
DEX                   counteract previous extra
                      INX
LDY #&FF
BMI check
```

By incrementing the X register we point to the next byte after the end of command marker. Normally this will be the first ASCII character in the next command of the command table. However, after the last command we can place a &00 byte to mark the end-of-command-table; if this is found the routine will exit. The DEX is to balance the extra INX issued earlier.

Once a command has been identified, control must be handed to the part of the language or routine responsible for carrying out the processes invoked by the command. This address is called the command's *execution address*. There are a number of ways in which this handing over of control can be performed – I'll describe the one I prefer to use, and then examine the others.

Basically I like to follow the command with its execution address – the command table is then built up as follows:

⟨COMMAND⟨address⟩⟩⟨COMMAND⟨address⟩⟩⟨COMMAND⟨address⟩⟩. . .

As an example, let's suppose our command table supports three commands called SOFT, LOUD, BEEP, and

the execution addresses for each are &2000, &2100 and &2200 respectively: the command table will look like that illustrated in figure 3.2.

TABLE

S	O	F	T	20	00	L	O	U	D	21	00	B	E	E	P	22	00

Figure 3.2 How the command table is stored in memory.

The problem here is that we need a way to mark the end of an individual command in the command table. One way would be to continue using an &FF byte and follow this by a two-byte address. This would be acceptable, but would waste a byte per command.

Notice that, for the time being, all our command tables are going to be within the standard user memory of the BBC Micro or Electron. This means that execution addresses will always be in the range &E00 to &7C00 (&6000 for Electron) though the upper limit is never likely to be reached. Now, addresses below &8000 have one thing in common – the most significant bit of the high-byte is always clear. For example, look at the bit pattern for the bytes &FF, &80, and contrast them with those for &7C, &30, and &0E (or any you care to choose) listed in figure 3.3. If the byte is less than &80 its top bit ie, the left-most is clear.

Byte	Bit pattern
&FF	1111 1111
&80	1000 0000
&7C	0111 1100
&30	0011 0000
&0E	0000 1110

Fig. 3.3: Bit patterns

If this, most significant, bit is set, the byte becomes a negative byte and the BMI and BPL instructions can be used effectively. Now for the crunch: as we know that

this top bit is positive, the high-byte can be used to mark the end of the command by setting the top bit and, when found, the byte and the one following it can be extracted. As the top bit of the high-byte should be clear to get the correct address it must then be stripped off before reforming the execution address and then jumping to it.

The command table used above becomes that shown in figure 3.4. The three negative bytes are the high bytes of each command's execution address; with &80 added they become &A0, &A1 and &A2 respectively. The &FF now solely marks the end of the command table.

TABLE

S	O	F	T	20+80	00	L	O	U	D	21+80	00	B	E	E	P	22+80	00	FF

Figure 3.4 The new command table, with negative bits.

The assembly language to perform all this is surprisingly simple. The only 'difficult' part left to do is to strip the negative (topmost) bit from the high-byte of the command execution address. This can be done by masking the high-byte by logically ANDing it with &7F, ie AND &7F. The new interpretive coding is as follows:

```
LDX #&FF            \ set X
LDY  #&FF           \ set Y
.check
INX                 \ increment both X and Y
INY
LDA (program),Y     \ get byte from command
CMP table,X         \ is it same as in command
                      table?
BEQ check           \ yes, repeat again
LDA table,X         \ get byte from command
                      table
CMP #&FF            \ is it end marker?
BEQ notfound        \ end reached -
unrecognised
BMI found           \ negative; so must be
                      address
```

```
.moveon          \ not this command - maybe
                   next one?
INX              \ move index on
LDA table,X      \ get next byte from table
BPL moveon       \ repeat until  negative: if
                   negative, start of next
                   command coming up
INX              \ move X to low byte
LDY #&FF         \ reset Y
BMI check        \ force jump
```

The extra INX in the *moveon* routine is to move the index on so that it is pointing at the low-byte of what is a two-byte address. The Y register is loaded with &FF before forcing a branch to *check* where both index registers will be incremented on to point to their respective first bytes for testing.

The *found* routine is responsible for extracting, masking and jumping to the command's execution address which is placed into the vector *addr*. On entry the accumulator contains the high-byte of the execution address:

```
.found
AND #&7F         \ mask top bit
STA addr+1       \ save high byte
INX              \ point index at low byte
LDA table,X      \ get low byte
STA addr         \ save low byte
JMP (addr)       \ indirect jump to execution
                   address
```

An interpretive call

The next program demonstrates the techniques above. It allows six commands to be called by pressing function key f1. Enter the listing as shown and save it. RUN the program and correct any errors that are flagged. When the program runs without error there will be a very slight pause while the program checks itself to see if any mistakes that are more subtle have been made.

All being well, the prompt will re-appear and you can save the final version of the interpreter program under the filename INT#1. If a checksum error is indicated, you'll need to check the assembly listing very carefully – check also that the checksum number

in line 1810 has been entered correctly.

Once the program has been run you can use the extra commands by pressing function key f1. This will perform CALL &900, at which stage an asterisk will be prompted. You can enter any of the commands you want here and press RETURN – the command will then be carried out. The commands are:

BEEP – sounds a bell character from the speaker
LOUD – makes BEEP loud
SOFT – makes BEEP quiet
SLOW – slows down the keyboard auto-repeat rate
NORM – restores the keyboard auto-repeat rate
HELP – lists the commands available

If you enter a command not known to the interpreter an 'Unknown command!' error message will be printed.

You can see the routines outlined in the text so far in the following lines:

routine	lines
check	380–470
moveon	490–550
not found	570–640
found	660–720
table	770–840

The other routines in the program are described briefly below.

routine	lines	action
start	180–340	Display * and read line of text using OSWORD 0
negative	860–890	Display error message
beep	910–940	Beeps internal speaker
soft	960–1060	Softens beep
loud	1020–1060	Restores loud beep
slow	1080–1120	Slows auto-repeat rate
norm	1140–1180	Restores normal auto repeat rate
help	1200–1290	Lists 'extra' commands
details	1310–1390	Extra commands printed by HELP

As you can see, the extra commands which have been added are basically all OSBYTE calls. You might like to try adding a few of your own at this stage. All you need to do is choose your command action, add the chosen name to the command table (at line 835, for example) and then add the extra assembler within the table; say, in lines 1291 to 1299. Renumbering will allow you access to more line numbers. Don't forget to add the new command *details* to the 'details' list.

If you're having trouble thinking of a new command try adding a one called HOME. This can 'home' the cursor to the top of the screen using CHR$ (30), and in assembly language it would be:

```
.home
LDA #30
JSR &FFEE
RTS
```

Finally the functions and procedures used in the program are detailed below.

FN/PROC	lines	action
FNaddress	1440–1580	Assembles command string and execution address into table
FNhelp	1600–1640	Assembles command string and a return character
FNequs	1660–1690	Performs EQUS
FNequb	1710–1740	Performs EQUB
PROCchecksum	1760–1840	Validates machine code

The routine *FNaddress* may need a little explanation. This has the name of the command passed through to it – the function immediately converts a copy of the command into lower-case letters, thus "BEEP" is converted into 'beep', "LOUD" into 'loud' and so on. This lower-case representation is the label used to mark the entry point to that particular routine. The final action of the function is to assemble the command string, followed by the value assigned to the label, into the command table.

Inserting the wedge

The program above, although showing that the interpreter worked, is not practical because we have to issue a command before we can enter the command required. By using the wedge approach outlined in Chapter 2 we can intercept the command line interpreter vector (CLIV): the wedge interpreter can test to see if it recognises the command. If it does, it executes the command; otherwise the interpreter restores all registers and jumps to the default CLIV address. Listing 3.2 is a hybrid of listing 2.1 and listing 3.1 so you should be able to recognise most of its routines and understand what their purpose is. The only real difference is in the construction of the command table. The command strings are now all prefixed by an asterisk because they are all accessed via a * command. On entry to the wedge, note that the index registers point to the first character of the command string, which will always be a *.

The program includes a checksum routine which will tell you if there are any errors in the code. Don't forget to save a copy when you have typed it all in – use "WEDGE2" as a filename. The machine-code for the wedge assembles at &5000 but this can be altered to suit your own needs. The wedge is automatically installed when the program is run, so all the commands are instantly available for use. The commands are as described for listing 3.1. To use any, just type the command, ie,

```
*BEEP
```

Note that *HELP will be trapped by the wedge – if you wish to use the MOS version of *HELP use *H. instead. Alternatively, restore all registers and jump to the old CLIV address after processing the wedge version of *HELP.

As with all wedges installed in this way, pressing the BREAK key will reset the CLIV, meaning that the revector routine must be called each time this happens, to redirect CLIV. This can be done best by calling the assembly address, ie,

```
CALL &5000
```

Of course you could save the machine-code and simply *RUN this each time it is needed. The machine code for listing 2.2 can be saved by:

```
*SAVE  EXTENS  5000 5134
```

More advanced readers could try setting the BRK jump vector with *FX247, *FX248 and *FX249 (see page 81).

A token gesture

As I mentioned earlier in this chapter, there are several ways in which a command can be terminated as well as giving access to an execution address. While the address method is very common, another popular approach is to use a *token*. A token is a single byte conveying both items of information: end-of-command; and the execution or action address of the command.

Marking the end of the command string is simple enough and is in the same vein – set the top bit of the token so it becomes a negative byte and can be identified by testing the negative flag of the status register. The command execution addresses must now be stored in a separate table of command execution addresses. If they are stored in an order corresponding to that of the command table entries the remaining 7 bits of the token byte can be used to provide an index into the command execution address table in the range 0 to 127. The command execution address table can itself be constructed in a number of ways: one way is to store all high-bytes together and all low-bytes together.

Using listing 3.2 as an example, the command table could be constructed thus:

```
.table
*BEEP" (&80+0)     \ negative bit plus 0 offset
*LOUD" (&80+1)     \ negative bit plus 1 offset
*SOFT" (&80+2)     \ negative bit plus 2 offset
*SLOW" (&80+3)     \ negative bit plus 3 offset
*NORM" (&80+4)     \ negative bit plus 4 offset
*HELP" (&80+5)     \ negative bit plus 5 offset
```

The &80 is used to produce the negative bit of the byte, &80 being 1000 0000 in binary. To this the offset value is added.

The command execution address tables can follow on along these lines:

```
highbytes
OPT   FNequb (beep DIV 256)
OPT   FNequb (loud DIV 256)
OPT   FNequb (soft DIV 256)
OPT   FNequb (slow DIV 256)
OPT   FNequb (norm DIV 256)
OPT   FNequb (help DIV 256)
```

The low-bytes are assembled in similar fashion:

```
lowbytes
OPT   FNequb (beep MOD 256)
OPT   FNequb (loud MOD 256)
OPT   FNequb (soft MOD 256)
OPT   FNequb (slow MOD 256)
OPT   FNequb (norm MOD 256)
OPT   FNequb (help MOD 256)
```

As the token will provide an exact index into the respective tables there's no need to use a marker byte to mark the end of the execution address tables. The end of the command tables should be marked in the normal fashion, though.

When a command has been identified, the token marking its end can be masked to erase the top negative bit. This leaves the offset which can be transferred into the X register and used as an index to extract the jump address:

```
.found
AND #&7F              \ mask top bit out
TAX                   \ move index into X
LDA highbytes,X       \ get high byte address
STA jumpvector+1      \ save in jump vector
LDA lowbytes,X        \ get low byte address
STA jumpvector        \ and save this
JMP (jumpvector)      \ do indirect jump to
                        command
```

Another approach would be to construct the command execution address table with both the address bytes stored side by side, low-byte followed by high-byte. The offset extracted from the token would then be a factor of two out. To correct for this, the offset must be multiplied by two. In machine code the instruction ASL A will do just this, provided that the offset is held within the accumulator. In such a case the *found* routine would become:

```
.found
AND #&7F              \ mask top bit out
ASL A                 \ multiply offset by 2
TAX                   \ move index into X
LDA address,X         \ get low-byte address
STA jumpvector+1      \ save in jump vector
INX                   \ increment index
LDA address,X         \ get high-byte
                        address
STA jumpvector        \ and save this
JMP (jumpvector)      \ do indirect jump to
                        command
```

All the above techniques can be improved, adapted and improvised upon in a number of ways, but the above methods are reasonably straightforward to follow, implement and use. It is well worth the time to master these techniques and make yourself familiar with their operation.

The programs so far have supported service interpreters: interpreters that will trap, recognise and execute a single command at a time. To be able to perform this task on a sequence of commands it is necessary to implement a language.

Listing 3.1

```
    REM PROG 3.1 - 14/3/86
10 REM Call Interpreter #1
20 REM Mastering Interpreters
30 REM and Compilers
40 REM (C) Bruce Smith 1986
50 :
60 PROCassemble
70 PROCchecksum
80 *KEY0 CALL start|M
90 END
```

```
10Ø :
11Ø DEF PROCassemble
12Ø param=&7Ø:buffer=&8Ø
13Ø osword=&FFF1:osasci=&FFE3
14Ø osbyte=&FFF4
15Ø FOR pass=Ø TO 3 STEP 3
16Ø P%=&9ØØ
17Ø [OPT pass
18Ø .start
19Ø LDA #ASC"*"
2ØØ JSR osasci
21Ø LDA #buffer MOD 256
22Ø STA param
23Ø LDA #buffer DIV 256
24Ø STA param+1
25Ø LDA #5
26Ø STA param+2
27Ø LDA #ASC"A"
28Ø STA param+3
29Ø LDA #ASC"Z"
3ØØ STA param+4
31Ø LDA #Ø
32Ø LDX #param
33Ø TAY
34Ø JSR osword
35Ø \
36Ø LDX #&FF
37Ø LDY #&FF
38Ø .check
39Ø INY
4ØØ INX
41Ø LDA buffer,Y
42Ø CMP table,X
43Ø BEQ check
44Ø LDA table,X
45Ø CMP #&FF
46Ø BEQ notfound
47Ø BMI found
48Ø \
49Ø .moveon
5ØØ INX
51Ø LDA table,X
52Ø BPL moveon
```

```
53Ø INX
54Ø LDY #&FF
55Ø BMI check
56Ø \
57Ø .notfound
58Ø LDY #&FF
59Ø .noloop
6ØØ INY
61Ø LDA negative,Y
62Ø BEQ out
63Ø JSR osasci
64Ø BPL noloop
65Ø \
66Ø .found
67Ø AND #&7F
68Ø STA param+1
69Ø INX
7ØØ LDA table,X
71Ø STA param
72Ø JMP (param)
73Ø \
74Ø .out
75Ø RTS
76Ø \
77Ø .table
78Ø OPT FNaddress("BEEP")
79Ø OPT FNaddress("LOUD")
8ØØ OPT FNaddress("SOFT")
81Ø OPT FNaddress("SLOW")
82Ø OPT FNaddress("NORM")
83Ø OPT FNaddress("HELP")
84Ø OPT FNequb(&FF)
85Ø \
86Ø .negative
87Ø OPT FNhelp("Unknown command!")
88Ø OPT FNequb(7)
89Ø BRK
9ØØ \
91Ø .beep
92Ø LDA #7
93Ø JSR osasci
94Ø RTS
95Ø \
```

```
 96Ø .soft
 97Ø LDA #212
 98Ø LDX #216
 99Ø JSR osbyte
1ØØØ RTS
1Ø1Ø \
1Ø2Ø .loud
1Ø3Ø LDA #212
1Ø4Ø LDX #144
1Ø5Ø JSR osbyte
1Ø6Ø RTS
1Ø7Ø \
1Ø8Ø .slow
1Ø9Ø LDA #11
11ØØ LDX #128
111Ø JSR osbyte
112Ø RTS
113Ø \
114Ø .norm
115Ø LDA #11
116Ø LDX #32
117Ø JSR osbyte
118Ø RTS
119Ø \
12ØØ .help
121Ø LDX #&FF
122Ø .helploop
123Ø INX
124Ø LDA details,X
125Ø BEQ done
126Ø JSR osasci
127Ø BPL helploop
128Ø .done
129Ø RTS
13ØØ \
131Ø .details
132Ø OPT FNhelp("Extensions 1.ØØ")
133Ø OPT FNhelp("BEEP")
134Ø OPT FNhelp("HELP")
135Ø OPT FNhelp("LOUD")
136Ø OPT FNhelp("NORM")
137Ø OPT FNhelp("SLOW")
138Ø OPT FNhelp("SOFT")
```

```
139Ø OPT FNequb(Ø)
14ØØ ]
141Ø NEXT
142Ø ENDPROC
143Ø :
144Ø DEF FNaddress (A$)
145Ø B$="":length%=LEN(A$)
146Ø FOR N%=1 TO length%
147Ø C$=MID$(A$,N%,1)
148Ø B$=B$+CHR$(ASC(C$)+32)
149Ø NEXT
15ØØ M%=EVAL(B$)
151Ø M%=(M% DIV 256)+&8Ø
152Ø N%=EVAL(B$) MOD 256
153Ø [OPT pass
154Ø OPT FNequs (A$)
155Ø OPT FNequb (M%)
156Ø OPT FNequb (N%)
157Ø ]
158Ø =pass
159Ø :
16ØØ DEF FNhelp(A$)
161Ø $P%=A$
162Ø P%=P%+LEN(A$)
163Ø ?P%=13:P%=P%+1
164Ø =pass
165Ø :
166Ø DEF FNequs (A$)
167Ø $P%=A$
168Ø P%=P%+LEN(A$)
169Ø =pass
17ØØ :
171Ø DEF FNequb (B%)
172Ø ?P%=B%
173Ø P%=P%+1
174Ø =pass
175Ø :
176Ø DEF PROCchecksum
177Ø X%=Ø
178Ø FOR N%=&9ØØ TO &9F9
179Ø X%=X%+?N%
18ØØ NEXT
181Ø IF X%=29361 THEN ENDPROC
```

```
182Ø VDU 7
183Ø PRINT "Assembler error!"
184Ø STOP
```

Listing 3.2

```
      REM PROG3.2 - 14/3/86
 1Ø REM Re-vectoring OSCLI
 2Ø REM Mastering Interpreters
 3Ø REM and Compilers
 4Ø REM (C) Bruce Smith 1986
 5Ø :
 6Ø PROCassemble
 7Ø PROCchecksum
 8Ø CALL revector
 9Ø PRINT "Extensions installed!"
1ØØ END
11Ø :
12Ø DEF PROCassemble
13Ø FOR pass=Ø TO 3 STEP 3
14Ø P%=&5ØØØ
15Ø [OPT pass
16Ø .revector
17Ø LDA &2Ø9
18Ø BPL error
19Ø STA clihigh
2ØØ LDA &2Ø8
21Ø STA clilow
22Ø LDA #wedge DIV 256
23Ø STA &2Ø9
24Ø LDA #wedge MOD 256
25Ø STA &2Ø8
26Ø RTS
27Ø \
28Ø .clilow
29Ø BRK
3ØØ .clihigh
31Ø BRK
32Ø \
33Ø .error
34Ø EQUW Ø
35Ø EQUS"Wedge already present!"
```

```
36Ø EQUB 7
37Ø EQUB Ø
38Ø \
39Ø .accum
4ØØ BRK
41Ø .xreg
42Ø BRK
43Ø .yreg
44Ø BRK
45Ø \
46Ø .wedge
47Ø STA accum
48Ø STX xreg
49Ø STX &7Ø
5ØØ STY yreg
51Ø STY &71
52Ø \
53Ø LDX #&FF
54Ø LDY #&FF
55Ø .check
56Ø INY
57Ø INX
58Ø LDA (&7Ø),Y
59Ø CMP table,X
6ØØ BEQ check
61Ø LDA table,X
62Ø CMP #&FF
63Ø BEQ notfound
64Ø BMI found
65Ø \
66Ø .moveon
67Ø INX
68Ø LDA table,X
69Ø BPL moveon
7ØØ INX
71Ø LDY #&FF
72Ø BMI check
73Ø \
74Ø .notfound
75Ø LDA accum
76Ø LDX xreg
77Ø LDY yreg
78Ø JMP (clilow)
```

```
 79Ø \
 8ØØ .found
 81Ø AND #&7F
 82Ø STA &71
 83Ø INX
 84Ø LDA table,X
 85Ø STA &7Ø
 86Ø JMP (&7Ø)
 87Ø \
 88Ø .table
 89Ø EQUS"*BEEP"
 9ØØ EQUB (beep DIV 256)+&8Ø
 91Ø EQUB beep MOD 256
 92Ø EQUS"*LOUD"
 93Ø EQUB (loud DIV 256)+&8Ø
 94Ø EQUB (loud MOD 256)
 95Ø EQUS"*SOFT"
 96Ø EQUB (soft DIV 256)+&8Ø
 97Ø EQUB (soft MOD 256)
 98Ø EQUS"*SLOW"
 99Ø EQUB (slow DIV 256)+&8Ø
1ØØØ EQUB slow MOD 256
1Ø1Ø EQUS"*NORM"
1Ø2Ø EQUB (norm DIV 256)+&8Ø
1Ø3Ø EQUB norm MOD 256
1Ø4Ø EQUS"*HELP"
1Ø5Ø EQUB (help DIV 256)+&8Ø
1Ø6Ø EQUB help MOD 256
1Ø7Ø EQUB &FF
1Ø8Ø \
1Ø9Ø .negative
11ØØ EQUS"Unknown command!"
111Ø EQUB 7
112Ø EQUB 13
113Ø EQUB Ø
114Ø \
115Ø .beep
116Ø LDA #7
117Ø JSR &FFE3
118Ø RTS
119Ø \
12ØØ .soft
121Ø LDA #212
```

```
122Ø LDX #216
123Ø JSR &FFF4
124Ø RTS
125Ø \
126Ø .loud
127Ø LDA #212
128Ø LDX #144
129Ø JSR &FFF4
13ØØ RTS
131Ø \
132Ø .slow
133Ø LDA #11
134Ø LDX #128
135Ø JSR &FFF4
136Ø RTS
137Ø \
138Ø .norm
139Ø LDA #11
14ØØ LDX #32
141Ø JSR &FFF4
142Ø RTS
143Ø \
144Ø \
145Ø .help
146Ø LDX #&FF
147Ø .helploop
148Ø INx
149Ø LDA details,X
15ØØ BEQ done
151Ø JSR &FFE3
152Ø BPL helploop
153Ø .done
154Ø LDA accum
155Ø LDX xreg
156Ø LDY yreg
157Ø JMP (clilow)
158Ø \
159Ø .details
16ØØ EQUB 13
161Ø EQUS"Command Extensions 1.ØØ"
162Ø EQUB 13
163Ø EQUS"BEEP"
164Ø EQUB 13
```

```
165Ø EQUS"HELP"
166Ø EQUB 13
167Ø EQUS"LOUD"
168Ø EQUB 13
169Ø EQUS"NORM"
17ØØ EQUB 13
171Ø EQUS"SLOW"
172Ø EQUB 13
173Ø EQUS"SOFT"
174Ø EQUB 13
175Ø EQUB Ø
176Ø ]
177Ø NEXT
178Ø ENDPROC
179Ø :
18ØØ DEF PROCchecksum
181Ø X%=Ø
182Ø FOR N%=&5ØØØ TO &5133
183Ø X%=X%+?N%
184Ø NEXT
185Ø IF X%=32596 THEN ENDPROC
186Ø VDU 7
187Ø PRINT "Assembler error!"
188Ø STOP
```

CHAPTER FOUR

GRAFRITE – THE LANGUAGE

This chapter presents a simple interpretive language that I have called Grafrite. In essence it is a graphics language and it is written in such a way that, as your expertise develops, you will find it easy to extend and add to. Indeed, chapters 5 and 6 present a variety of routines which could all be used to develop the language, as well as giving a set of 'standard' conversion subroutines that are listed in Appendix A.

When the idea of a language presents itself, it is vitally important that you spend some time sorting out exactly what it is you want your language to do. This may seem self-evident but some hard thinking now will help enormously when you come to sit down at the keyboard and type in the first lines of code. The time you should spend on this phase of development is hard to quantify – only you can tell when you have reached it. For instance, I personally like to mull over ideas for at least a couple of weeks. You will need to make notes as you decide on the approach to be taken and these should be jotted down in a notebook dedicated to your language.

My own guidelines for developing an interpretive language for this book were:

- The source and object code must fit into a standard BBC Model B computer, typically with default PAGE at &1900 (thus letting it fit into a Master and Electron).
- The interpreter core should be capable of supporting almost any extension.
- The language coding itself should be minimal.
- The language should support variables.
- The language should support a simple loop structure.

The first point seems obvious but, as we shall see in Chapter 7, it's possible to assemble very long programs that would not normally fit into a standard BBC micro or Electron. Also bear in mind that in addition to the space required to assemble the source code, BASIC also needs its own workspace to store variables and values above TOP.

On its completion the Grafrite source program is a little over 10k in length, and can generate and coexist with some 1300 bytes of machine code. This just runs in the available memory while maintaining sensible but meaningful variable names.

By keeping the machine-code performing the actions required of the language short and sweet we can concentrate on the action and implementation of the interpreter itself. The MOS allows all standard graphics to be produced using simple VDU codes. These are generated by loading the accumulator with a code number and calling OSWRCH.

Tools

As with most tasks in life, having the correct tools before you start means a better working environment, which will give you a better end result and, most importantly, keep you sane! What are those tools? The most important development tools are pen and paper, closely followed by a machine-code monitor program – I am assuming you already have a computer! Before you approach the computer, pen and paper are used to design your program, flowchart its operation, design the memory map and write the main routines. Invest in a hard-bound book of the sort used for cash analysis – this keeps all your notes in one place and saves having sheets of paper all over the desk.

The machine-code monitor should ideally be ROM-based and will prove invaluable when it comes to debugging your program at the testing stage which will be discussed later. If you don't have such a ROM, consult magazine reviews to help you choose. All offer much the same in the way of facilities, so make your selection on ease of use – you don't want to spend all your time learning how to use the firmware; in this respect, the BBC Monitor from BBC Publications scores highly.

Planning

To me the planning stage is the most interesting part of writing any long program. The first thing is to make a list of the actions you want the program to take. These fall into two categories, the *foreground* and *background* tasks. The foreground tasks are the ones you see on screen; for example, the printing of the start-up screen, and the reading in of a line of text, the command entered being either acted upon or stored within the program storage area. Background tasks are the ones which are unseen but essential to the program's operation. These would include clearing and setting up memory pointers, transferring a line of input to the program area, and searching the command table for a command.

Books often advise using a traditional flowchart, but most programmers I know tend to draw these up after they've written the program! The method I prefer is to write a sort of English pseudo-code: essentially, write a list of the program's actions in the order they are to be executed. This can be done by dividing a sheet of paper up into three columns, heading each in turn *label*, *action*, *comments*. The *action* column is where I place my simple description as to what is to happen. The first column, *label*, is used to mark jumps, and the *comments* column is for me to add any afterthoughts. Figure 4.1 shows my pseudo-code flowchart for Grafrite. Note that *gpage* and *gtop* are the Grafrite equivalents of BASIC's PAGE and TOP.

label	action	comments
start	set up pointers	ie,gpage, next free location
	set gtop	use &FF
	clear all variables	need prefix to variable
	clear and display	name is Grafrite!
main	display prompt	use ok⟩
	read line of text into input buffer	use OSWORD 0
	?immediate;goto *im*	ie,LIST,RUN *
	copy from input buffer to program area	
	reset pointer	ie, gtop
	goto *main*	
im	if LIST goto *list*	lists program
	if NEW goto *new*	deletes program

	if RUN goto *run*	starts program execution
	if * then do OS	pass to OSCLI
	END	
list	print program	finishes when reaches &FF
	goto *main*	
new	ask if sure	print prompt read key
	If Y goto *start*	complete rest
	goto *main*	abort
run	Search Command Table	
	found extract address	stored after command
	JMP (address)	
	Not found do error	print command
	goto *main*	
command	do command action	
	reset interpreter	
	End of program?	
	No goto *run*	
	Yes goto *main*	

Figure 4.1: Pseudo-code flowchart.

This figure shows that I have only incorporated four command actions for the interpreter, ie RUN, LIST, NEW and *. The last one, *, is to give access to all operating system commands. These commands are known at the start as they form part of what I originally defined as the *core interpreter* in the list of objectives above.

The next step is to make a list of provisional commands that the interpreter will support. This list should be kept as small as practical during the implementation stage, the ideal number of commands being that which will take you to implement the main features of the language. As the interpreter will be dealing with standard BBC BASIC graphics commands, the temptation is to use commands such as MODE, DRAW, and so on which, though perfectly acceptable, seems to defeat the object of writing the interpreter. The commands I chose were ones that reflected the operation while trying to keep away from BBC BASIC as much as possible. The commands and their actions are listed in table 4.1 below.

Command	Action
DLINE	Draws dotted line

DOT	Plots a single dot
LINE	Draws a line
MOVE	Moves graphics cursor
REPEAT	Starts loop
SCREEN	Selects a screen mode
TRIANGLE	Draws and fills a triangle
WIPE	Clears the graphics screen
WHILE	Loop control command

Table 4.1: Provisional commands

Only MOVE and REPEAT bear any resemblance to BBC BASIC commands.

The memory map

Grafrite (and any other language you may write) can claim the language workspace allocated by the MOS. The language workspace is split into three sections. The first is zero page locations &00 to &8F, the second is the block from &400 to &7FF. Finally there is the space from OSHWM to HIMEM, which in a Mode 7 disc system would normally be &1900 to &7BFF (&1D00 to &6000 for a Mode 6 Electron-Plus 3).

There is one large drawback at this stage, however. If you use a monitor to help you design and develop your program it will immediately claim some (if not all) of the language workspace for its operation. So, when planning the memory map of your language make sure you have a list of the locations used by the monitor itself and use only those that it leaves untouched. If you are in any doubt, only use what you know to be safe locations. I prefer not to touch the language workspace, and note only the locations normally available until I have a tried, tested and debugged interpreter core. You should follow the same route and only then should you claim use of the language workspace. Figure 4.2 contains a list of these free usable locations and their usage.

Address	Normal Use
&70–&8F	Free user memory
&90–&9F	Econet workspace
&800–&83F	Sound workspace
&840–&87F	Sound channel buffers

&880–&8BF	Printer buffer
&8C0–&8FF	Envelope 1–4 storage
&900–&9BF	Envelope 5–16 storage or RS423 output buffer
&9C0–&9FF	Speech buffer
&900–&9FF	Cassette output buffer
&A00–&AFF	RS423 or cassette input buffer
&B00–&BFF	Function key buffer (Econet or Master series)
&C00–&CFF	Font character storage (Econet or Master series)

Figure 4.2: 'Free' RAM areas

The use of these 'free' areas, of course, depends on their not being needed by their normal occupants while the program is being used. For example, use of Econet workspace must be avoided if a network is the filing system. Use of the function key buffer should be a last resort only – inadvertently pressing a function key could cause most peculiar effects!

The Grafrite listing at the end of this chapter uses some of the above 'free' locations and will run on all current Acorn micros. At this stage of planning it is not vital to define the workspace for the finished version, but it is worth thinking about during development.

It is now worth sketching out your planned development memory map. Again, fancy diagrams are nice but a simple clear list is just as effective and easier to alter as you adapt the program or perhaps re-think your memory map. Figure 4.3 outlines the Grafrite memory map.

Address	Function
&70–&8F	Vectors and storage
&900–&9FF	Variable area
&C00–&C80	Input buffer
&CF0–&CFF	Number buffer
PAGE-HIMEM	Grafrite programs
HIMEM–&3000	Grafrite interpreter
&3000–&7FFF	Screen memory

Figure 4.3: Grafrite memory map

Vectors and storage

The use of vectors was discussed in chapter two. To any 6502 machine-code programmer, vectors are immensely important and this is particularly true when it comes to memory management – a vital point in writing languages. Programming in languages like BASIC is easy. You don't have to worry about where your program is because that's usually not really important. However, suppose you have a very long program which stretches into the 20k screen memory boundary (in other words, GTOP (ie, Grafrite TOP) extends beyond &3000). When running the program BASIC encounters a change from the current mode to Mode 2 and interprets this, but then checks to see where TOP is. Seeing that TOP is greater than the proposed change of HIMEM it aborts and prints a Bad Mode error message. If it did not perform this check the mode change would result in the top of your program's being cut off from its base and lost forever; and the chances are this would result in an almost irreversible Bad Program error. The language must, therefore, be responsible for all memory management and for checking possible conflicts of interest.

In addition to the above, several bytes should be reserved for saving registers, the page value of OSHWM and a jump vector to allow indirect jumps to the command execution address. These are all standard requirements of languages and a definition of these might look like:

```
program = &70
progtop = &72
gpage = &74
Tstore = &75
Ystore = &76
Xstore = &77
jumpvector = &78
```

Let's look more closely at the function of each byte. The first two bytes labelled *program* (&70 and &71) simply point to the start of the program storage space, and these will be used by the interpreter when the program is run.

The next two bytes are more important. Labelled *progtop* (&72 and &73), they will always hold an

address which points to the first free byte at the top of the current Grafrite program. Each time a new command is added to the program it will be placed at this start address. Obviously the contents of *progtop* will have to be incremented as each new command is added, to ensure that its current contents always indicate the next free location.

The location at *gpage* (&74) simply holds the high-byte of OSHWM.

Tstore, *Xstore* and *Ystore* (&75, &76, and &77) are byte stores: the last two are for the X and Y index registers and the first is a temporary byte store, free for any register which needs it at that moment.

Finally, the two bytes at *jumpvector* will generally contain an address through which indirect jumps can be made. In principle this should be kept sacred to the interpreter proper.

Variables

The structure of variables, ie, characters which may have changing numeric values assigned to them, must be given thought at this point. In general, variables may be of four types: single-byte, double-byte, four-byte and five-byte.

Single-byte variables are limited to numbers in the range 0 to 255. This is clearly restrictive to any language, especially a graphics language which should be able to address co-ordinates on the screen in the range 0 to 1279.

Double-byte variables may contain values in the range 0 to 65535. Four-byte values cater for huge values, both positive and negative. Five-byte numbers are reserved to process floating point values. Of these four types, two-byte values are the easiest to cope with and would fit neatly into our list of requirements.

Variables are normally distinguished in some way to enable the interpreter to readily recognise them and therefore need the minimum of coding for this purpose. For example, integer variables in BBC BASIC are postfixed by a % sign. It is much easier, though, to *prefix* a symbol to a variable. This means that a variable can be identified simply by testing its first character. In Grafrite, variables will be prefixed by a hash, '#'.

The next decision at this stage is whether variables should be static or dynamic.

Static variables are similar to BBC BASIC's integer variables A% to Z% – they are each assigned specific bytes of memory.

Dynamic variables are ones that are added to the program as needed and normally have names assigned to them by the user. In BBC BASIC variables such as *test%* and *counter%* would be examples of dynamic integer variables. The location of these in memory will vary as the BASIC interpreter assigns a space above the program when the program's run.

I'll stick to static variables for now as they're are much easier to manipulate; but I'll discuss dynamic variables in Chapter 7 when we look at ways of improving the Grafrite interpreter. Variables will also be limited to a single character in the range A to Z inclusive, ie, #A, through to #Z, giving 26 two-byte variables in total.

Command syntax

Now that the main decisions have been made, it's possible to sit down and work out the syntax the language will use. This means deciding what your interpreter is going to accept as input and whether a RUN command has been issued. The pseudo-code flowchart (page 43) shows that we first need to distinguish between two types of command input:

First is the *immediate* command. This is one that the interpreter will act on but not add to the program storage space as soon as it is recognised. Examples are LIST and RUN. Immediate commands can be identified by looking through a command table containing a list of them. However, if the command is not recognised as an immediate command, control must pass to the main line-handling routine that then adds the command to the *program* storage space.

Table 4.1 shows that all of the proposed commands, except REPEAT and WIPE, need some kind of information to act upon. Except for SCREEN and WHILE, all commands require a pair of graphics co-ordinates. SCREEN needs a screen mode value, and WHILE a condition that it can evaluate. This information can be passed in two ways: first, *directly*, by specifying it after the command as a number or variable, thus:

LINE 8200,&500 or LINE #X,#Y

or second, *indirectly*, by coding the command so that it will extract the data from a predefined memory location. For instance, all commands needing graphics co-ordinates could simply extract them from, say, the memory locations assigned to the variables #X and #Y. The variables could be defined before the command is used:

```
#X = &&200
#Y = &&500
LINE
```

LINE could now draw to the point &200,&500 on the screen.

Grafrite uses both direct and indirect methods. The indirect method is used for co-ordinates as described above, while the direct method is used for SCREEN and WHILE. SCREEN expects a numeric value and WHILE a variable.

The structure of Grafrite's only *control loop* is next on the list. Taking it at face value, the loop will REPEAT WHILE a variable is true. Designer's licence allows you to make your own choice here. For example, I have decided that Grafrite will REPEAT the loop WHILE the variable is not zero. When the specified variable reaches zero the loop will terminate.

In Grafrite programming terms the following loop would execute 10 times:

```
#A = &10
#B = &1
REPEAT
#A = #A - #B
WHILE #A
```

Let's now look at the syntax of variable assignments. These fall into three categories, and the following list shows how they can be defined.

Numeric assignment	:	var = numeric value
Equality	:	var = var
Operator	:	var = var ⟨ + − * / ⟩ var/numeric (ie, add, subtract, multiply, or divide)

A numeric assignment in BBC BASIC would be something like,

A% = 255 or A% = &FF

To keep the length of the source program to a minimum, Grafrite only accepts hexadecimal values for numeric assignment (Appendix B contains the assembly code for ASCII hex-to-binary conversion should you wish to use this instead). The correct syntax in Grafrite then is:

#A = &FF

A typical two-byte numeric assignment would be

#B = &CAFE

Assigning the value of one variable to another is straightforward if left as a simple ⟨var⟩ = ⟨var⟩, typically:

#A = #B

Use of operators is a little more complex. Again Grafrite is limited in this respect, making use of just + and − operators which must be confined to acting on variables – ie adding one variable to another or subtracting one from another. The accepted way to write this would be:

#A = #A + #B

Here we are saying 'let variable A be made equal to the current contents of variable A plus the contents of variable B'. Similarly the expression:

#X = #X − #S

reads 'let the variable X be made equal to the current contents of variable X minus the contents of variable S'. The expression itself can be simplified by getting rid of the first, assignment, part of it. For example:

#A = #A + #B becomes #A + #B
#X = #X − #S becomes #X − #S

In both cases the first variable is adjusted by the contents of the second variable in a manner defined by the sign.

The program, the errors

The structure of a Grafrite program is outlined next, along with the sort of error messages it might generate. Line numbers will not be used. Instead, program commands will be added at *progtop* as and when they are entered into the micro. Each program command will be terminated by a single carriage return – this will have benefits when we LIST. The program top will be marked by &FF.

Error messages are important, and can help with smooth development of the language. They should be generated by either the interpreter or the command's execution code. The command interpreter will normally only generate an error when it fails to identify a command. A typical error message might be:

'Command not known'

Other errors will usually be ones of syntax and as such are handled by the command execution code. This is the segment of the program to which the command interpreter passes control once it has identified the command. For example, failure to specify a screen mode number after SCREEN or a control variable after WHILE requires the interpreter to inform the user of this. Similarly an illegal variable such as #! must also be checked for. Each of these will be discussed in more detail as we construct the Grafrite source program.

Writing Grafrite

I use a 'top down' approach when it comes to writing programs. For Grafrite this means starting with the screen display and set-up routines, next the interpreter; and then, when all that is working satisfactorily, I finally add the command execution code.

The first decision is where the source program is a) going to sit and b) where the object code will be assembled. To allow easy editing of the program, the two should ideally sit side-by-side in memory. On a disc-based BBC Micro, PAGE is usually set at &1900 – unless, of course, you have some extra ROMs installed that may have bumped up the value of PAGE. Standard disc accesses such as LOAD and SAVE can be performed quite happily using the bare minimum of the disc workspace. For example, assembling machine code at &1200 will have no harmful effects when it

comes to a LOAD or SAVE. This means that the source program can be entered at the default value of PAGE and the machine-code assembled with P% set to &1200. If the object code becomes corrupted all that's needed is to re-run the program, since we have the source program to hand (and, ideally, saved to disc or tape before each RUN).

If you are using the Advanced Disc Filing System (ADFS) (as supplied on the BBC B+, Master or Electron Plus 3) you can still proceed in this way – however, the ADFS normally stores a copy of the current disc catalogue in its workspace. Obviously this will be corrupted by Grafrite object code, and cause an error when you next try to access a disc if you assemble code here. This problem can be corrected by simply typing *MOUNT (which will re-load the disc catalogue).

Owners of Masters and tape-based systems will usually have PAGE set to &E00, so they should firstly set PAGE to &1900, ie:

```
PAGE = &1900
NEW
```

and carry on as described below. Of course, any time you press the BREAK key you will need to reset the value of PAGE to get back to your program, ie:

```
PAGE = &1900
OLD
```

This is best done be re-defining the BREAK key as a function key by entering:

```
*KEY10 PAGE = &1900|M  OLD|M
```

You could also make this the first line of the Grafrite source program.

Once the assembly language program has been completed the various variables and pointers can be adjusted to make use of the main memory before the final object code is assembled and saved, as is detailed later in the chapter.

Taking the above approach means that it is absolutely vital that you use variables to hold addresses and pointers. When it comes to adjusting addresses all that's needed is to alter the variables. If you don't do this, you'll have to manually change every location

reference in the program. Therefore:

LDA (program), Y

is preferable to:

LDA (&70), Y

Initialisation

The first thing that happens when a language is selected, eg, BASIC, is that it displays a message to tell you what it is. Grafrite displays the sign-on message:

Acorn MOS
Grafrite Graphics

which involves printing out an ASCII string of characters on the screen – reasonably straightforward. However, we will need to print various ASCII character strings while writing the language, ie, error messages and prompts. What would be nice is a universal print routine capable of printing any ASCII string. The problem here is telling the printing routine exactly *where* the string is – and the easiest way is to place the address of the string on the stack. This can be done best by performing a JSR to the print routine and following it with the ASCII string to be printed:

```
JSR stringprint     \ call routine pushing
                      address
EQUW &0C0D          \ clear screen and perform
                      return
EQUS "Acorn MOS"
EQUW &0D0D          \ two returns
EQUS "Grafrite Graphics"
EQUW &0D0D          \ two more returns
NOP                 \ do nothing
```

The routine is rather devious and is listed below. The RTS address from the *stringprint* routine is now on the stack – this is extracted and placed into a vector. Using indirect indexed addressing the bytes after the JSR *stringprint* are extracted and printed until a NOP instruction is encountered (this marks the end of the data to be printed). At this time the vector address points to one after the NOP instruction and so an indirect jump through the vector can be made back into the program.

The choice of NOP is not significant in itself; but what is, is its opcode of &EA: the most significant bit is already set and can therefore be tested with the BMI instruction! The *stringprint* routine is detailed below:

```
.stringprint
PLA                 \ get low byte address from
                      stack
STA lowsave         \ place in vector address
PLA                 \ get high byte address from
                      stack
STA highsave        \ place in vector address
LDY #0              \ clear indirect index
.mainprint
INC lowsave         \ add one to low byte address
BNE noup            \ branch if not equal
INC highsave        \ else add one to high byte
.noup
LDA (lowsave),Y     \ get ASCII character
BMI donestring      \ branch if a negative byte
JSR osasci          \ print the character
BPL mainprint       \ do next byte
.donestring
JMP (lowsave)       \ do indirect jump back
```

Getting back to the first command after the NOP instruction is done by performing an indirect jump through *lowsave*.

The next step in the initialisation process is to set all program variables to zero. This is done by writing a zero into each byte. There are 26 two-byte variables in all: therefore, the first 52 (&35) bytes of memory from *varbase* need to be cleared, where *varbase* marks the start of the variable storage area:

```
.clear
LDX #&35            \ set index
LDA #0              \ clearing byte
.clearvars
STA varbase,X       \ clear byte
DEX                 \ decrement index
BPL clearvars       \ continue until X<0
```

The last thing to do is to set up any vectors and place &FF to mark the top of the program – effectively, put this at the start of the program storage space since we have no program at present.

The main vector in use is *progtop*. The variable *start* can be used to hold the program start address. The development version of Grafrite uses memory from &1800 upward as the program storage area. This can be adjusted to suit your own requirements and is chosen because it sits just above the Grafrite object code. Take care not to enter a very long source program during the development stage without first saving the Grafrite assembly listing, as the Grafrite Language program could go beyond &1900 and corrupt the Grafrite source listing.

```
.setup
LDA #0              \ low byte is 0
STA progtop
LDA #start          \ high byte in 'start'
STA progtop+1
STA page            \ set page as well
```

Because we only need the high byte of the program area, *start* should only be set to this, ie:

start = &18

The protocol for setting the top of program is a return character (&0D) followed by &FF.

```
.setGTOP
LDY #0              \ clear indirect index
LDA #&0D            \ ASCII return
STA (progtop),Y     \ place in memory
INY                 \ increment index
LDA #&FF            \ GTOP marker
STA (program),Y     \ set GTOP
```

GTOP has been defined to mean the Grafrite TOP, to avoid confusion with BASIC's pseudo-variable TOP.

Prompting input

Now that the initialisation process is complete Grafrite should be ready to accept input from the user. To signal this, a prompt should be printed. The Grafrite prompt is:

ok〉

which is done simply by printing the ASCII characters involved:

```
.out
LDA #ASC("o")
JSR oswrch
LDA #ASC("k")
JSR oswrch
LDA #ASC(">")
JSR oswrch
```

With this done Grafrite is ready to receive information from the keyboard. This is readily available by using an OSWORD call with the accumulator containing 0.

As with all OSWORD calls the two index registers should point to a parameter block in memory which contains the information listed in Table 4.2.

Offset from start of parameter blk	Contents
0	low-byte of input buffer address
1	high-byte of input buffer address
2	maximum length of input line
3	smallest ASCII value acceptable
4	largest ASCII value acceptable

Table 4.2: OSWORD 0 parameter block

The input buffer address is the address the line being typed-in will be written to: &C00. The maximum line length is the number of characters the line input routine can accept. To keep this within page &C of memory, 255, the maximum, is chosen. Finally, the range of ASCII characters acceptable must be defined. The chosen range should obviously be those characters that are also acceptable to the Grafrite interpreter. In most cases this is the space character (&20) through to the Delete character (&7F). Table 4.3 shows how this information needs to be arranged in the parameter block.

Offset	Contents	Note
0	&00	low-byte input buffer address
1	&0C	high-byte input buffer address
2	&FF	maximum length of input line
3	&20	smallest ASCII value acceptable (Space character)
4	&7F	largest ASCII value acceptable (Delete character)

Table 4.3: Contents of the parameter block

The assembly code for this is straightforward enough:

```
.readline
LDA #ipbuffer MOD 256       \ low byte of buffer
                              address
STA pblock                  \ store at 0
LDA #ipbuffer DIV 256       \ high byte of buffer
                              address
STA pblock+1                \ store at 1
LDA #linelen                \ get acceptable
                              line length
STA pblock+2                \ store at 2
LDA #minasc                 \ get minimum
                              character
STA pblock+3                \ store at 3
LDA #maxasc                 \ get maximum
                              character
STA pblock+4                \ store at 4
LDX #pblock MOD 256         \ low byte of pblock
LDY #pblock DIV 256         \ high byte of pblock
LDA #0                      \ call code
JSR osword
```

The one item not defined above was just where in memory the parameter block was to be, as pointed to by the variable *pblock*. I chose to use the space above the Grafrite variables in page &9, from &940, but it can be defined as you want.

When keyboard input is expected by a program it's absolutely vital that the routine tests to see if the ESCAPE key has been pressed. If it has, it must be acknowledged with an OSBYTE call or the micro will

just hang up. The question is how to detect this. When using OSWORD 0 the carry flag is used to signal this condition. If it is set, ie, C = 1, then ESCAPE has been pressed, otherwise we are free to continue. BCS *error* will detect the condition, and the ESCAPE acknowledgement would go something like this:

```
.error
LDA #126              \ Escape acknowledge code
JSR osbyte
JSR stringprint       \ print an error message
EQUW &0007            \ return and bell
EQUS "<<Escape>>"
EQUB &0D
NOP                   \ negative byte
JMP out               \ display prompt
```

Because ESCAPE has been detected we can ignore the line entered previously (if any) and return to the prompt-printing section of the program and start again.

Line by line

The next procedure is to locate just where the desired input text is within the input buffer. This is not as odd as it may seem: for example, several spaces may have been typed before the actual command was entered. The presence or absence of spaces should not generate an error, unless a space is used to delimit a command from some following parameters. By ignoring them at this stage space can be saved when it comes to transferring commands to GTOP. Note that the line could consist of just a carriage return, which should also be tested for and, if found, also ignored.

```
.moveup               \ subroutine to find
                        non-space character
LDY #&FF              \ set index ready for
                        incrementing
.pastloop
INY                   \ bump index
LDA ipbuffer,Y        \ get byte from input buffer
CMP #&20              \ is it a space?
BEQ pastloop          \ yes: move to next character
CMP #&0D              \ is it a return?
```

```
BEQ itscr          \ yes: so branch
CLC                \ signal input present
RTS                \ return to calling routine
.itscr
SEC                \ signal no input
RTS                \ return to calling routine
```

Once again the carry flag has been used to indicate a condition. In this instance if the carry flag is set (C = 1), there is nothing in the input buffer; so the prompt should be displayed once more and the keyboard read line performed again. If, however, the carry is clear (C = 0), action can move on to the interpreter. The calling routine for *moveup* would be structured thus:

```
JSR moveup         \ move past spaces
BCS out            \ if return detected go to
                    'out'

LIST
NEW
RUN
*
```

At this point the Y register contains the offset into the input buffer to the first byte in the command and needs to be preserved for future operations.

Immediate action

The first part of the interpretive process is ready to begin. The language must test to see whether the line entered contains an *immediate* command – that is, one that should be acted upon right away. There are four commands that Grafrite will evaluate as immediate:

LIST
NEW
RUN
*

The command table for these is constructed: each is marked off with a token value, and the table is postfixed with an &FF byte.

```
.immediate          \ immediate command table
EQUS "LIST"
EQUB &80            \ list token
EQUS "NEW"
EQUB &81            \ new token
EQUS "RUN"
EQUB &82            \ run token
EQUS "*"
EQUB &83            \ star token
EQUB &FF            \ end of command table
```

A simple 'compare and test' interpreter similar to those described in the last chapter is then used to try and identify the command.

```
.testlist
STY Ystore          \ preserve Y
DEY                 \ decrement Y
LDX #&FF            \ use X as index into command
                      table
.listloop
INY                 \ bump Y
INX                 \ bump X
LDA immediate,X     \ get byte from command table
CMP ipbuffer,Y      \ is it same as input buffer?
BEQ listloop        \ repeat if same
LDA listloop,X      \ reset flags in status
                      register
BMI token           \ if negative then it's a
                      token
.moveon             \ if not, move to next command
INX                 \ bump X
LDA immediate,X     \ get byte from command table
BPL moveon          \ continue while ASCII
LDY Ystore          \ reset Y to point to command
DEY                 \ decrement Y
JMP listloop        \ and try next command
```

As mentioned earlier, it is vital that the contents of the Y register are preserved so that the start of the command under scrutiny in the input buffer can always be found by using the Y as an index in from *ipbuffer*. This is performed at the start of the *testlist* routine by storing it

in *Ystore*. While Y is being used to step through the bytes of the command in the input buffer, X is used to access the bytes in the command table, and is set to &FF before entering *listloop* at which stage both index registers are incremented by one.

Listing 4.1

```
10 REM Grafrite Graphics
20 REM Mastering Compilers
30 REM and Interpreters
40 REM (C) Bruce Smith 1986
50 :
60 PROCvariables
70 PROCassemble
80 PROCchecksum
90 END
100 :
110 DEF PROCvariables
120 osbyte=&FFF4
130 osrdch=&FFE0
140 osasci=&FFE3
150 osnewl=&FFE7
160 oswrch=&FFEE
170 osword=&FFF1
180 varbase=&900
190 xbase=&92E
200 pamblk=&940
210 pageval=&1800
220 ipbuffer=&C00
230 Hbuffer=&CF0
240 start=&18
250 program=&70
260 progtop=&72
270 page=&74
280 Tstore=&75
290 Ystore=&76
300 Xstore=&77
310 mode=&78
320 temp=&79
330 lowsave=&80
340 highsave=&81
350 jumpvector=&82
360 hexstore=&84
```

```
37Ø repeatjump=&86
38Ø M%=&FF:L%=&2Ø:H%=&8Ø
39Ø ENDPROC
4ØØ :
41Ø DEF PROCassemble
42Ø FOR PASS=Ø TO 3 STEP 3
43Ø P%=&12ØØ
44Ø [OPT PASS
45Ø .entry
46Ø JSR stringprint
47Ø OPT FNequw (&ØCØD)
48Ø OPT FNequs ("Acorn MOS")
49Ø OPT FNequw (&ØDØD)
5ØØ OPT FNequs ("Grafrite Graphi
51Ø OPT FNequw (&ØDØD)
52Ø NOP
53Ø \
54Ø .clear
55Ø LDX #&33
56Ø LDA #Ø
57Ø \
58Ø .clearvars
59Ø STA varbase,X
6ØØ DEX
61Ø BPL clearvars
62Ø \
63Ø LDA #Ø
64Ø STA progtop
65Ø LDA #start
66Ø STA progtop+1
67Ø STA page
68Ø LDY #Ø
69Ø LDA #13
7ØØ STA (progtop),Y
71Ø INY
72Ø LDA #255
73Ø STA (progtop),Y
74Ø \
75Ø .out
76Ø LDA #ASC"o"
77Ø JSR oswrch
78Ø LDA #ASC"k"
79Ø JSR oswrch
```

```
 8ØØ LDA #ASC">"
 81Ø JSR oswrch
 82Ø \
 83Ø LDA #ipbuffer MOD 256
 84Ø STA pamblk
 85Ø LDA #ipbuffer DIV 256
 86Ø STA pamblk+1
 87Ø LDA #M%
 88Ø STA pamblk+2
 89Ø LDA #L%
 9ØØ STA pamblk+3
 91Ø LDA #H%
 92Ø STA pamblk+4
 93Ø LDA #Ø
 94Ø LDX #pamblk MOD 256
 95Ø LDY #pamblk DIV 256
 96Ø JSR osword
 97Ø BCC noerror
 98Ø JMP error
 99Ø \
1ØØØ .noerror
1Ø1Ø JSR moveup
1Ø2Ø BCS out
1Ø3Ø JSR testlist
1Ø4Ø BCS out
1Ø5Ø JSR copyline
1Ø6Ø JMP out
1Ø7Ø \
1Ø8Ø .copyline
1Ø9Ø DEY
11ØØ TYA:TAX
111Ø LDY #255
112Ø \
113Ø .moveloop
114Ø INY:INX
115Ø LDA ipbuffer,X
116Ø STA (progtop),Y
117Ø CMP #13
118Ø BNE moveloop
119Ø INY
12ØØ \
121Ø .movedone
122Ø CLC:TYA
```

```
123Ø ADC progtop
124Ø STA progtop
125Ø BCC nohigh
126Ø INC progtop+1
127Ø \
128Ø .nohigh
129Ø LDY #Ø
13ØØ LDA #255
131Ø STA (progtop),Y
132Ø RTS
133Ø \
134Ø .testlist
135Ø STY Ystore
136Ø DEY
137Ø LDX #&FF
138Ø STX Xstore
139Ø \
14ØØ .listloop
141Ø INY:INX
142Ø LDA immediate,X
143Ø CMP ipbuffer,Y
144Ø BEQ listloop
145Ø LDA immediate,X
146Ø BMI token
147Ø \
148Ø .moveon
149Ø INX
15ØØ LDA immediate,X
151Ø BPL moveon
152Ø LDY Ystore
153Ø DEY
154Ø JMP listloop
155Ø \
156Ø .token
157Ø CMP #&FF
158Ø BEQ notimmed
159Ø CMP #&8Ø
16ØØ BEQ dolist
161Ø CMP #&81
162Ø BEQ donew
163Ø CMP #&82
164Ø BNE over
165Ø JMP dorun
```

```
166Ø .over
167Ø CMP #&83
168Ø BNE notimmed
169Ø JMP doMOS
17ØØ \
171Ø .notimmed
172Ø LDY Ystore
173Ø CLC:RTS
174Ø \
175Ø .dolist
176Ø JSR osnewl
177Ø LDA page
178Ø STA temp+1
179Ø LDA #1
18ØØ STA temp
181Ø LDY #Ø
182Ø LDA (temp),Y
183Ø CMP #&FF
184Ø BEQ noprogram
185Ø DEY
186Ø DEC temp
187Ø \
188Ø .printlist
189Ø INY
19ØØ LDA (temp),Y
191Ø BMI donelist
192Ø JSR osasci
193Ø CPY #&FF
194Ø BNE printlist
195Ø INC temp+1
196Ø BNE printlist
197Ø \
198Ø .donelist
199Ø JSR osnewl
2ØØØ SEC:RTS
2Ø1Ø \
2Ø2Ø .noprogram
2Ø3Ø JSR stringprint
2Ø4Ø OPT FNequs ("No program present")
2Ø5Ø OPT FNequw (&ØDØ7)
2Ø6Ø NOP
2Ø7Ø SEC:RTS
2Ø8Ø \
```

```
2090 .donew
2100 JSR stringprint
2110 OPT FNequs ("Are you sure? (Y/N)")
2120 NOP
2130 JSR osrdch
2140 CMP #ASC"Y"
2150 BNE notsure
2160 LDA #22
2170 JSR oswrch
2180 LDA #7
2190 JSR oswrch
2200 PLA:PLA
2210 JMP entry
2220 \
2230 .notsure
2240 JSR osnewl
2250 JMP dolist
2260 \
2270 .dorun
2280 PLA:PLA
2290 JMP interpreter
2300 \
2310 .doMOS
2320 LDX #ipbuffer MOD 256
2330 LDY #ipbuffer DIV 256
2340 JSR &FFF7
2350 SEC:RTS
2360 \
2370 .moveup
2380 LDY #255
2390 .pastloop
2400 INY
2410 LDA ipbuffer,Y
2420 CMP #32
2430 BEQ pastloop
2440 CMP #13
2450 BEQ itscr
2460 CLC:RTS
2470 \
2480 .itscr
2490 SEC:RTS
2500 \
2510 .stringprint
```

```
252Ø PLA
253Ø STA lowsave
254Ø PLA
255Ø STA highsave
256Ø LDY #Ø
257Ø \
258Ø .mainprint
259Ø INC lowsave
26ØØ BNE noup
261Ø INC highsave
262Ø .noup
263Ø LDA (lowsave),Y
264Ø BMI donestring
265Ø JSR osasci
266Ø BPL mainprint
267Ø .donestring
268Ø JMP (lowsave)
269Ø \
27ØØ .error
271Ø LDA #126
272Ø JSR &FFF4
273Ø JSR stringprint
274Ø OPT FNequw (&ØDØ7)
275Ø OPT FNequs ("<Escape>")
276Ø OPT FNequb (13)
277Ø NOP
278Ø JMP out
279Ø \
28ØØ .immediate
281Ø OPT FNequs ("LIST")
282Ø OPT FNequb (&8Ø)
283Ø OPT FNequs ("NEW")
284Ø OPT FNequb (&81)
285Ø OPT FNequs ("RUN")
286Ø OPT FNequb (&82)
287Ø OPT FNequs ("*")
288Ø OPT FNequb (&83)
289Ø OPT FNequb (&FF)
29ØØ \
291Ø .commands
292Ø OPT FNaddress ("DLINE")
293Ø OPT FNaddress ("DOT")
294Ø OPT FNaddress ("LINE")
```

```
295Ø OPT FNaddress ("MOVE")
296Ø OPT FNaddress ("REPEAT")
297Ø OPT FNaddress ("SCREEN")
298Ø OPT FNaddress ("TRIANGLE")
299Ø OPT FNaddress ("WHILE")
3ØØØ OPT FNaddress ("WIPE")
3Ø1Ø OPT FNequb (&FF)
3Ø2Ø \
3Ø3Ø .interpreter
3Ø4Ø LDX #&FF
3Ø5Ø LDY #&FF
3Ø6Ø LDA page
3Ø7Ø STA program+1
3Ø8Ø LDA #Ø
3Ø9Ø STA program
31ØØ \
311Ø INY:INY
312Ø LDA (program),Y
313Ø CMP #&FF
314Ø BEQ end
315Ø DEY:DEY
316Ø \
317Ø .identify
318Ø INY:INX
319Ø LDA (program),Y
32ØØ CMP #ASC"#"
321Ø BEQ variable
322Ø AND #&DF
323Ø CMP commands,X
324Ø BEQ identify
325Ø LDA commands,X
326Ø BMI address
327Ø \
328Ø .moveon
329Ø INX
33ØØ LDA commands,X
331Ø BPL moveon
332Ø LDY #&FF
333Ø INX
334Ø JMP identify
335Ø \
336Ø .end
337Ø JMP out
```

```
338Ø \
339Ø .address
34ØØ CMP #&FF
341Ø BEQ notknown
342Ø AND #&7F
343Ø STA jumpvector+1
344Ø INX
345Ø LDA commands,X
346Ø STA jumpvector
347Ø INY
348Ø STY Ystore
349Ø JMP (jumpvector)
35ØØ \
351Ø .notknown
352Ø JSR stringprint
353Ø OPT FNequs ("Don't know ")
354Ø NOP
355Ø \
356Ø .details
357Ø LDY #&FF
358Ø .queryloop
359Ø INY
36ØØ LDA (program),Y
361Ø JSR oswrch
362Ø CMP #&D
363Ø BNE queryloop
364Ø JSR osnewl
365Ø JMP out
366Ø \
367Ø .resetinterp
368Ø LDX #&FF
369Ø LDY #Ø
37ØØ LDA (program),Y
371Ø CMP #&FF
372Ø BEQ end
373Ø DEY
374Ø JMP identify
375Ø \
376Ø .variable
377Ø JSR chrcon
378Ø STA Xstore
379Ø INY
38ØØ LDA(program),Y
```

```
381Ø CMP #ASC"-"
382Ø BNE tryadd
383Ø INY
384Ø LDA(program),Y
385Ø CMP #ASC"#"
386Ø BEQ syok1
387Ø JMP syntax
388Ø \
389Ø .syok1
39ØØ JSR sub1
391Ø SEC
392Ø SBC varbase,X
393Ø JSR sub2
394Ø SBC varbase,X
395Ø JSR sub3
396Ø JMP resetinterp
397Ø \
398Ø .tryadd
399Ø CMP #ASC"+"
4ØØØ BNE tryequals
4Ø1Ø INY
4Ø2Ø LDA(program),Y
4Ø3Ø CMP #ASC"#"
4Ø4Ø BEQ synok2
4Ø5Ø JMP syntax
4Ø6Ø \
4Ø7Ø .synok2
4Ø8Ø JSR sub1
4Ø9Ø CLC
41ØØ ADC varbase,X
411Ø JSR sub2
412Ø ADC varbase,X
413Ø JSR sub3
414Ø JMP resetinterp
415Ø \
416Ø .tryequals
417Ø CMP #ASC"="
418Ø BNE syntax
419Ø INY
42ØØ LDA(program),Y
421Ø CMP #ASC"#"
422Ø BNE trynumber
423Ø JSRchrcon
```

```
424Ø STA Tstore
425Ø LDX Tstore
426Ø LDA varbase,X
427Ø LDX Xstore
428Ø STA varbase,X
429Ø INC Xstore
43ØØ INC Tstore
431Ø LDX Tstore
432Ø LDA varbase,X
433Ø LDX Xstore
434Ø STA varbase,X
435Ø STY Ystore
436Ø JSR endadd
437Ø JMP resetinterp
438Ø \
439Ø .trynumber
44ØØ CMP #ASC"&"
441Ø BNE syntax
442Ø TYA:PHA
443Ø JSR movestring
444Ø PLA
445Ø STA Ystore
446Ø JSR endadd
447Ø JSR asctobin
448Ø LDA hexstore
449Ø LDX Xstore
45ØØ STA varbase,X
451Ø LDA hexstore+1
452Ø INX
453Ø STA varbase,X
454Ø JMP resetinterp
455Ø \
456Ø .sub1
457Ø JSR chrcon
458Ø STA Tstore
459Ø LDX Xstore
46ØØ LDA varbase,X
461Ø LDX Tstore
462Ø RTS
463Ø \
464Ø .sub2
465Ø LDX Xstore
466Ø STA varbase,X
```

```
467Ø INC Xstore
468Ø INC Tstore
469Ø LDX Xstore
47ØØ LDA varbase,X
471Ø LDX Tstore
472Ø RTS
473Ø \
474Ø .sub3
475Ø LDX Xstore
476Ø STA varbase,X
477Ø STY Ystore
478Ø JSR endadd
479Ø RTS
48ØØ \
481Ø .syntax
482Ø JSR stringprint
483Ø OPT FNequs ("Syntax error at ")
484Ø NOP
485Ø JMP details
486Ø \
487Ø .chrcon INY
488Ø LDA(program),Y
489Ø SEC
49ØØ SBC #&41
491Ø ASL A
492Ø CMP #Ø
493Ø BCC badvar
494Ø CMP #&34
495Ø BCS badvar
496Ø RTS
497Ø \
498Ø .badvar
499Ø JSR stringprint
5ØØØ OPT FNequs ("Bad variable at ")
5Ø1Ø NOP
5Ø2Ø JMP details
5Ø3Ø \
5Ø4Ø .screen
5Ø5Ø DEY
5Ø6Ø JSR movepast
5Ø7Ø BCC getmode
5Ø8Ø JMP screenerror
5Ø9Ø \
```

```
51ØØ .getmode
511Ø LDA (program),Y
512Ø SEC
513Ø SBC #&3Ø
514Ø STA mode
515Ø INY
516Ø STY Ystore
517Ø \
518Ø LDA #22
519Ø JSR oswrch
52ØØ LDA mode
521Ø JSR oswrch
522Ø JSR endadd
523Ø JMP resetinterp
524Ø \
525Ø .move
526Ø JSR endadd
527Ø LDA #4
528Ø JMP doplot
529Ø \
53ØØ .line
531Ø JSR endadd
532Ø LDA #5
533Ø JMP doplot
534Ø \
535Ø .dot
536Ø JSR endadd
537Ø LDA #69
538Ø JMP doplot
539Ø \
54ØØ .dline
541Ø JSR endadd
542Ø LDA #21
543Ø JMP doplot
544Ø \
545Ø .triangle
546Ø JSR endadd
547Ø LDA #85
548Ø \
549Ø .doplot
55ØØ PHA
551Ø LDA #25
552Ø JSR oswrch
```

```
553Ø PLA
554Ø JSR oswrch
555Ø STX Xstore
556Ø LDX #Ø
557Ø \
558Ø .plotloop
559Ø LDA xbase,X
56ØØ JSR oswrch
561Ø INX
562Ø CPX #4
563Ø BNE plotloop
564Ø LDX Xstore
565Ø JMP resetinterp
566Ø \
567Ø .wipe
568Ø JSR endadd
569Ø LDA #16
57ØØ JSR oswrch
571Ø JMP resetinterp
572Ø \
573Ø .repeat
574Ø DEY
575Ø LDA (program),Y
576Ø CMP #&D
577Ø BEQ doadd
578Ø INY
579Ø BNE repeat
58ØØ \
581Ø .doadd
582Ø INY
583Ø STY Ystore
584Ø JSR addwhy
585Ø LDA program
586Ø STA repeatjump
587Ø LDA program+1
588Ø STA repeatjump+1
589Ø JMP resetinterp
59ØØ \
591Ø .while
592Ø DEY
593Ø \
594Ø .dowhile
595Ø INY
```

```
596Ø LDA (program),Y
597Ø CMP #&D
598Ø BNE tryhash
599Ø JMP novar
6ØØØ \
6Ø1Ø .tryhash
6Ø2Ø CMP #ASC"#"
6Ø3Ø BNE dowhile
6Ø4Ø JSR chrcon
6Ø5Ø TAX
6Ø6Ø LDA varbase,X
6Ø7Ø BNE andagain
6Ø8Ø LDA varbase+1,X
6Ø9Ø BEQ complete
61ØØ \
611Ø .andagain
612Ø LDA repeatjump
613Ø STA program
614Ø LDA repeatjump+1
615Ø STA program+1
616Ø JMP resetinterp
617Ø \
618Ø .complete
619Ø STA repeatjump+1
62ØØ JSR endadd
621Ø JMP resetinterp
622Ø \
623Ø .novar
624Ø JSR stringprint
625Ø OPT FNequs ("Control variable at ")
626Ø NOP
627Ø JMP details
628Ø \
629Ø .incprog
63ØØ CLC
631Ø LDA program
632Ø ADC #1
633Ø STA program
634Ø BCC noinchi
635Ø INC program+1
636Ø .noinchi
637Ø RTS
638Ø \
```

```
639Ø .subprog
64ØØ SEC
641Ø LDA program
642Ø SBC #1
643Ø STA program
644Ø BCS nodechi
645Ø DEC program+1
646Ø .nodechi
647Ø RTS
648Ø \
649Ø .addwhy
65ØØ LDA Ystore
651Ø CLC
652Ø ADC program
653Ø STA program
654Ø BCC noinchi
655Ø INC program+1
656Ø RTS
657Ø \
658Ø .movepast
659Ø LDA (program),Y
66ØØ CMP #&2Ø
661Ø BNE notspace
662Ø INY
663Ø BNE movepast
664Ø \
665Ø .notspace
666Ø CMP #&ØD
667Ø BNE noCR
668Ø SEC:RTS
669Ø \
67ØØ .noCR
671Ø CLC:RTS
672Ø \
673Ø .screenerror
674Ø JSR stringprint
675Ø OPT FNequs ("No mode at ")
676Ø NOP
677Ø JMP details
678Ø \
679Ø .endadd
68ØØ LDY Ystore
681Ø DEY
```

```
682Ø \
683Ø .ealoop
684Ø LDA (program),Y
685Ø INY
686Ø CMP #&ØD
687Ø BNE ealoop
688Ø STY Ystore
689Ø JMP addwhy
69ØØ \
691Ø .movestring
692Ø LDX #Ø
693Ø INY
694Ø LDA(program),Y
695Ø CMP #&2Ø
696Ø BEQ movestring
697Ø CMP #&D
698Ø BNE transfer
699Ø JMP numerror
7ØØØ \
7Ø1Ø .transfer
7Ø2Ø LDA (program),Y
7Ø3Ø STA Hbuffer,X
7Ø4Ø INY:INX
7Ø5Ø CPX #5
7Ø6Ø BNE transfer
7Ø7Ø RTS
7Ø8Ø \
7Ø9Ø .numerror
71ØØ JSR stringprint
711Ø OPT FNequs ("Bad hex at ")
712Ø NOP
713Ø JMP details
714Ø \
715Ø .asctobin
716Ø LDY #Ø
717Ø LDX #hexstore
718Ø STY Ø,X
719Ø STY 1,X
72ØØ STY 2,X
721Ø \
722Ø .next
723Ø LDA Hbuffer,Y
724Ø JSR ascbin
```

```
725Ø BCS bad
726Ø ASL A:ASL A
727Ø ASL A:ASL A
728Ø STY 2,X
729Ø LDY #4
73ØØ \
731Ø .ragain
732Ø ASL A
733Ø ROL Ø,X
734Ø ROL 1,X
735Ø DEY
736Ø BNE ragain
737Ø LDY 2,X
738Ø INY
739Ø BNE next
74ØØ \
741Ø .bad
742Ø LDA 2,X
743Ø RTS
744Ø \
745Ø .ascbin
746Ø CMP #48
747Ø BCC illegal
748Ø CMP #58
749Ø BCC undernine
75ØØ SBC #7
751Ø BCC illegal
752Ø CMP #64
753Ø BCS return
754Ø \
755Ø .undernine
756Ø AND #&ØF
757Ø .return
758Ø RTS
759Ø .illegal
76ØØ SEC:RTS
761Ø ]
762Ø NEXT
763Ø ENDPROC
764Ø :
765Ø DEF FNaddress (A$)
766Ø B$="":length%=LEN(A$)
767Ø FOR N%=1 TO length%
```

```
768Ø C$=MID$(A$,N%,1)
769Ø B$=B$+CHR$(ASC(C$)+32)
77ØØ NEXT
771Ø M%=EVAL(B$)
772Ø M%=(M% DIV 256)+&8Ø
773Ø N%=EVAL(B$) MOD 256
774Ø [OPT PASS
775Ø OPT FNequs (A$)
776Ø OPT FNequb(M%)
777Ø OPT FNequb(N%)
778Ø ]
779Ø =PASS
78ØØ :
781Ø DEF FNequb (B%)
782Ø ?P%=B%:P%=P%+1
783Ø =PASS
784Ø :
785Ø DEF FNequw (W%)
786Ø ?P%=W% MOD 256
787Ø P%?1=W% DIV 256
788Ø P%=P%+2
789Ø =PASS
79ØØ :
791Ø DEF FNequs (A$)
792Ø $P%=A$
793Ø P%=P%+LEN(A$)
794Ø =PASS
795Ø :
796Ø DEF PROCchecksum
797Ø R%=Ø
798Ø FOR Q%=&12ØØ TO &171C
799Ø R%=R%+?Q%
8ØØØ NEXT
8Ø1Ø IF R%=145223 THEN ENDPROC
8Ø2Ø PRINT "Checksum error!"
8Ø3Ø STOP
```

CHAPTER FIVE

LANGUAGE? – WHAT LANGUAGE?

Having spent all of the last chapter explaining the inner workings of Grafrite, our interpretive language, I now confess (wait for it) – Grafrite is not a language! Well, not yet. Why? Consider what happens when you press BREAK – you return to BASIC. Press the BREAK key while in BASIC and what happens? You stay in BASIC. For Grafrite to fully make the transition from a fancy piece of machine code to a language it must react as BASIC does to a BREAK: in short, stay in Grafrite.

There is another aspect to consider. Grafrite is entered via a CALL command. CALL performs a JSR to the specified address, leaving an RTS address on the stack. A full-bodied language would not be expecting to perform an RTS and as such should itself take control of the stack.

From the debugging and testing point of view, it is much better to approach language writing as outlined already. The two changes above should be left till the last. Taking control of the BREAK key means you must also take control of the error-handling facilities. Use those already available from BASIC rather than waste time and effort supplying our own routines – which could themselves be wrong!

Handling BRK

Before tackling the BREAK key let us first examine how we can trap the software BRK. In BBC BASIC the BRK instruction (opcode &00) is used to define an error. For instance, when an unknown command occurs, BASIC jumps to a suitable error message, for example 'Mistake'. The error message is stored like this:

```
BRK                 \ software BRK
04                  \ error number
"Mistake"           \ ASCII message
BRK                 \ software BRK
```

To see this in action try the following short program:

Listing 5.1

```
10 REM Error Messages
20 REM Mastering Interpreters
30 REM and Compilers
40 REM (C) Bruce Smith 1986
50 :
60 P%=&9000
70 [
80 .error
90 LDA #7
100 JSR &FFE3
110 BRK
120 ]
130 ?P%=5
140 P%=P%+1
150 $P%="Not a Mistake!"
160 P%=P%+LEN($P%)
170 [
180 BRK
190 ]
200 CALL error
```

When you run this it will print out the error message 'Not A Mistake!' Try typing:

PRINT ERL

and the error number will be printed. So it does work. Before indirecting through BRKV the MOS does some housekeeping along these lines:

```
STA &FC             \ save accumulator
PLA                 \ set flags at BRK
PHA                 \ resave
AND #&10            \ was it an IRQ?
BNE brk             \ no: branch to BRK routine
JMP (&204)          \ jump through IRQ1V
.brk
TXA                 \ save X register
PHA                 \ by pushing to stack
```

```
TSX                 \ move stack pointer to X
LDA  &103,X         \ get low byte address from
                      stack
CLD                 \ clear decimal flag
SEC                 \ set carry flag
SBC #1              \ subtract 1
STA &FD             \ save low byte
LDA &104,X          \ get high byte
SBC #0              \ subtract carry
STA &FE             \ save high byte
```

On BRK the program counter address plus 1 (two bytes) followed by the status register are placed on the hardware stack, then the MOS is entered. Before pushing the status register the BRK flag in the register will have been set to indicate that a BRK has occurred. The processor then performs an indirect jump to &FFFE which, in turn, leads it into the MOS. Because &FFFE is also used by the processor when an Interrupt Request (IRQ) is encountered, the BRK handling code must test for this – by testing the BRK *flag* in the status register.

If the BRK flag is set the *brk* routine is followed; otherwise an indirect jump via IRQ1V is performed to the Interrupt Request handling code within the MOS.

The remainder of the routine above extracts the RTS address from the stack and 1 is subtracted and the contents stored in a vector at &FD,&FE. The reason for subtracting 1 is to make the vector point to the error code number. Using indirect addressing through the vector, the error message can be printed, which is what BASIC does. We can follow this format, but I have chosen to modify it slightly to:

Error code x ⟨message⟩

A simple print routine can display the first half of the message and the vector then used to print the error code and message. The following segment assumes that the error code is in the range 0 to 9.

```
.break
 LDY #&FF           \ set index
.errorloop
INY                 \ increment index
```

```
LDA mess,Y          \ get byte
BEQ over            \ go over if 0
JSR osasci          \ print it
BNE errorloop       \ do again
.mess               \ first part message
OPT FNequs ("Error code ")
OPT FNequb (0)
.over
LDY #0              \ set index
LDA (&FD),Y         \ get error code
CMP #1              \ was it 1? (my code for
                      ESCAPE)
BNE noescape        \ no branch
PHA                 \ else save error code
LDA #126            \ acknowledge ESCAPE
JSR osbyte          \ through MOS
PLA                 \ restore
.noescape
CLC
ADC #&30            \ convert to ASCII "0"-"9"
JSR osasci          \ print error number
.printloop
INY                 \ increment Y
LDA (&FD),Y         \ get byte
BEQ alldone         \ all done if 0
JSR osasci          \ print it
BNE printloop
.alldone
```

The error code in this instance is kept in the range 0 to 9. This makes it simple to convert the error number into an ASCII character for printing, just by adding &30 to the error number. The vector at &FD,&FE is used by the print routine to access the error code and the error message that comes after that.

To make use of the above BRK handling we must direct BRKV to point at the *break* routine above. This is done by altering the contents of the BRK vector, thus:

```
.in
SEI                 \ disable interrupts
PHP                 \ save status register
LDA #break MOD 25   \ get low byte
STA &202            \ save
```

```
LDA #break DIV 25 \ get high byte
STA &203          \ save
PLP               \ restore status
CLI               \ clear interrupts
```

After redirecting BRKV the 'language' should take control by performing three operations:

- clear the decimal flag
- enable interrupts
- re-set the stack

This is performed in just five bytes,

```
CLD               \ clear decimal flag
CLI               \ enable interrupts
LDX #&FF
TSX               \ set stack pointer to &FF
```

All this should be handled when the language is powered up; so now we could display the signing-on message. Error messages can also now be embedded within the text in the way described earlier. If Grafrite classes ESCAPE as error code number one, then the error message could be entered thus:

```
.escape
BRK
OPT FNequb (1)
OPT FNequs ("<Escape>")
BRK
```

Performing a branch or jump to 'escape' will result in the error message:

Error code 1 ⟨Escape⟩

being printed.

Handling BREAK

Now that BRK has been trapped let's turn our attention to the BREAK key itself. OSBYTE calls &F7, &F8 and &F9 allow us to point the MOS to a section of machine code which is to be executed when the BREAK key is pressed. In fact the three locations that these OSBYTE

calls write to, contain a machine-code instruction which is in effect a direct jump along the lines of:

JMP in

where *in* is the label marking the entry point – already detailed above. The opcode for JMP is &4C so this is written to the first byte with OSBYTE &F7. As with all OSBYTE calls, the byte to be written is held in the X register:

```
.grafrite
 LDX #in MOD 256     \low byte address
 LDA #&F8
 JSR osbyte          \ write with &F8
 LDX #in DIV 256     \ high byte address
 LDA #&F9
 JSR osbyte          \ write with &F9
 LDX #&4C            \ JMP opcode
 LDA #&F7
 JSR osbyte          \ write with &F7
 JMP in              \ continue
```

When the BREAK key is pressed the MOS will look to see if the byte written to by OSBYTE &F7 contains 0. If it does not it jumps to that location or else it continues along its normal route. The routine above need only be performed once when the language is entered. The JMP *in* instruction is the BRKV re-vectoring code.

Now each time BREAK is pressed Grafrite (or whatever language) will be re-entered. Even CTRL-BREAK will not have any effect other than to re-enter the language: turning the micro off will be the only way to restore normal conditions. However, I prefer to allow CTRL-BREAK to re-select the default language (normally BASIC). To do this we must rework the *in* coding (described above) slightly.

```
LDA #&FD
JSR osbyte          \ Do OSBYTE &FD
CPX #2              \ was it CTRL-BREAK?
BNE noe             \ no: so branch
LDX #0              \ yes:
```

```
LDA #&F7          \ so write 0 to reset
JSR osbyte        \ using MOS
RTS               \ crash back to BASIC
.noe
```

The OSBYTE call &FD can be used to read the status of the last BREAK. On return the X register will contain one of three numbers and will signify what type of BREAK last occurred:

0 = soft BREAK
1 = power-up reset
2 = hard BREAK

A 0 indicates that the BREAK key only was pressed; a 1 indicates that the machine has just been turned on; and a 2 indicates that a hard BREAK, i.e., CTRL-BREAK has occurred. If a 2 is encountered in the X register the JMP byte in the BREAK intercept vector (the bytes written to by OSBYTEs &F7, &F8 and &F9) must be reset to 0. If this does not happen then the next time you press the BREAK key you will re-enter Grafrite or the language in question!

Listing 5.2 shows how all the techniques described in this chapter can be strung together. Once you have played round with it you might like to have a go at working them into the main Grafrite listing.

The front-end of the program simulates Grafrite: pressing any key will cause it to be displayed on the screen, though pressing the ESCAPE, TAB or DELETE keys will generate an error. Pressing the BREAK key will clear the screen and re-enter Grafrite and CTRL-BREAK will allow you to leave the program.

The program itself assembles at &900 in under a page of memory. Table 5.1 indicates where you can find the routines discussed in this chapter; Table 5.2 outlines those not discussed.

Routine	Lines
grafrite	180 to 280
errorloop	330 to 380
over	400 to 480

notesc	500 to 530
printloop	550 to 600
alldone	620 to 720
in	740 to 900
noe	920 to 970

Table 5.1: routines described in text

Routine	Lines	Action
screen	990 to 1040	Display start-up message
doneit	1060 to 1170	Display prompt; read keyboard, Jump if error, else print key
escape	1190 to 1230	⟨ESCAPE⟩ error message
tab	1250 to 1290	⟨TAB⟩ error message
delete	1320 to 1350	⟨DELETE⟩ error message
startup	1370 to 1460	ASCII data

Table 5.2: routines not described in text

Listing 5.2

```
10 REM Intercepting BRK
20 REM Mastering Interpreters
30 REM and Compilers
40 REM (C) Bruce Smith 1986
50 :
60 osasci=&FFE3
70 osnewl=&FFE7
80 osbyte=&FFF4
90 osrdch=&FFE0
100 PROCassemble
110 PROCchecksum
120 CALL grafrite
130 END
140 :
150 DEF PROCassemble
160 FOR pass=0 TO 3 STEP 3
170 P%=&9000
180 [OPT pass
190 .grafrite
200 LDX #in MOD 256
210 LDA #&F8
```

```
22Ø JSR osbyte
23Ø LDX #in DIV 256
24Ø LDA #&F9
25Ø JSR osbyte
26Ø LDX #&4C
27Ø LDA #&F7
28Ø JSR osbyte
29Ø JMP in
3ØØ \
31Ø .break
32Ø LDY#255
33Ø \
34Ø .errorloop
35Ø INY
36Ø LDA mess,Y
37Ø BEQ over
38Ø JSR osasci
39Ø BNE errorloop
4ØØ \
41Ø .over
42Ø LDY #Ø
43Ø LDA (&FD),Y
44Ø CMP #1
45Ø BNE notesc
46Ø PHA
47Ø LDA #126
48Ø JSR osbyte
49Ø PLA
5ØØ \
51Ø .notesc
52Ø CLC
53Ø ADC #&3Ø
54Ø JSR &FFEE
55Ø \
56Ø .printloop
57Ø INY
58Ø LDA(&FD),Y
59Ø BEQ alldone
6ØØ JSR &FFEE
61Ø BNE printloop
62Ø \
63Ø .alldone
64Ø CLD
```

```
65Ø CLI
66Ø LDX #&FF
67Ø TXS
68Ø LDA #13
69Ø JSR osasci
7ØØ JMP doneit
71Ø .mess
72Ø OPT FNequs("Error code ")
73Ø OPT FNequb(Ø)
74Ø \
75Ø .in
76Ø PHP
77Ø SEI
78Ø LDA #break MOD 256
79Ø STA &2Ø2
8ØØ LDA #break DIV 256
81Ø STA &2Ø3
82Ø PLP
83Ø CLI
84Ø LDA #&FD
85Ø JSR osbyte
86Ø CPX #2
87Ø BNE noe
88Ø LDX #Ø
89Ø LDA #&F7
9ØØ JSR osbyte
91Ø RTS
92Ø \
93Ø .noe
94Ø CLI
95Ø CLD
96Ø LDX #&FF
97Ø TXS
98Ø LDX #&FF
99Ø \
1ØØØ .screen
1Ø1Ø INX
1Ø2Ø LDA startup,X
1Ø3Ø BEQ doneit
1Ø4Ø JSR osasci
1Ø5Ø BNE screen
1Ø6Ø \
1Ø7Ø .doneit
```

```
1Ø8Ø LDA #ASC"%"
1Ø9Ø JSR osasci
11ØØ JSR osrdch
111Ø BCS escape
112Ø CMP #9
113Ø BEQ tab
114Ø CMP #&7F
115Ø BEQ delete
116Ø JSR osasci
117Ø JSR osnewl
118Ø JMP doneit
119Ø \
12ØØ .escape
121Ø BRK
122Ø OPT FNequb(1)
123Ø OPT FNequs(" <Escape>")
124Ø BRK
125Ø \
126Ø .tab
127Ø BRK
128Ø OPT FNequb(2)
129Ø OPT FNequs(" <TAB>")
13ØØ BRK
131Ø \
132Ø .delete
133Ø BRK
134Ø OPT FNequb(3)
135Ø OPT FNequs(" <Delete>")
136Ø BRK
137Ø \
138Ø .startup
139Ø OPT FNequb(&C)
14ØØ OPT FNequb(&D)
141Ø OPT FNequs("Acorn MOS")
142Ø OPT FNequb(&D)
143Ø OPT FNequb(&D)
144Ø OPT FNequs("Grafrite Graphics")
145Ø OPT FNequb(&D)
146Ø OPT FNequb(&D)
147Ø OPT FNequb(Ø)
148Ø ]
149Ø NEXT
15ØØ ENDPROC
```

```
151Ø :
152Ø DEF FNequs(A$)
153Ø $P%=A$
154Ø P%=P%+LEN(A$)
155Ø =pass
156Ø :
157Ø DEF FNequb(byte%)
158Ø ?P%=byte%
159Ø P%=P%+1
16ØØ =pass
161Ø :
162Ø DEF PROCchecksum
163Ø T%=Ø
164Ø FOR L%=&9ØØ TO &9EB
165Ø T%=T%+?L%
166Ø NEXT
167Ø IF T%=27945 THEN ENDPROC
168Ø VDU 7
169Ø PRINT "Checksum error!"
```

CHAPTER SIX

IMPROVEMENTS

The Grafrite interpreter lacks a very necessary feature – the ability to load and save programs. That's not exactly true, of course, as the interpreter will recognise and allow use of * commands which means that *LOAD and *SAVE are both possible. To use both of these, though, the address of the program area must be known, and this is defined by the variable *start* in the Grafrite source code. If *start* is assigned a value of &1800 then a program can be loaded in with:

*LOAD program 1800

To save a program you need to specify a start and end address. The latter is subject to some guesswork unless you add a command to display this by printing the value from *progtop* which holds the value of GTOP. But in essence a *SAVE is simply:

*SAVE program 1800 xxxx

where xxxx is a suitable value.

Try entering a short program (one from chapter 4 would be suitable) and save it using an end address one page greater than the start address. If the start address is &1800 the end address would be &1900:

*SAVE program 1800 1900

Enter NEW (and respond Y for yes) then load the program; you'll find that you can list and run it in the normal way. But try adding a command to the end, and then LIST the program. You'll find that the program loaded has vanished and you are left with the command which you just entered. The reason this has happened is that the value of GTOP has *not been reset* to take account of the *LOAD – the Grafrite command interpreter still 'thinks' that GTOP is its default value. What is needed is a command to re-set GTOP, which is located at *progtop*.

Setting GTOP can be performed with a simple loop. As GTOP is marked by &FF all that's needed is for the program to search for this and load its address into *progtop* – performed like this:

```
.set
LDA #start          \ get start of program area
STA program+1       \ set it
LDA #0
STA program         \ and low byte
TAY                 \ index is 0
DEY                 \ decrement for loop entry
.findFF
INY                 \ bump index
LDA (program),Y     \ get byte from program
BMI foundFF         \ if negative then we've
                      found GTOP
CPY #&FF            \ is it maximum index?
BNE findFF          \ no: so repeat loop
INC program+1       \ yes: increment page
BNE findFF          \ and force branch
.foundFF
TYA                 \ move index into
                      accumulator
CLC
ADC program         \ add it to program low byte
STA program         \ and save the result
STA progtop         \ set low byte of GTOP
LDA program+1       \ get high byte
STA progtop+1       \ set high byte of GTOP
```

We need to insert a keyword into the command table to allow this 'new' command to be used. It is, of course, an immediate command and so needs to be inserted into the immediate command table. Any suitable name will do, and I have used SET in my version of Grafrite. You will need to re-arrange the table somewhat, as a direct branch to *set* will not be possible, ie:

```
CMP #&84        \ is it SET?
BNE notset      \ no: so skip JMP
JMP set         \ yes: so jump to set
.noset
```

The token &84 can be used to indicate SET, and assembled thus:

```
OPT FNequs ("SET")
OPT FNbyte (&84)
```

The SET command code can be located anywhere in the interpreter, but to keep things tidy I suggest you put it after the existing commands, but before the sub-routines, at the back end of the source code.

Once the Grafrite object code has been reassembled and saved the new command can be used. Its use is straightforward enough. *LOAD the program as already described above. Then type:

```
SET
```

To make sure that all is well, try typing in an extra command, then LIST the program. The command should have been added to the program listing.

Program editing

Sadly, Grafrite lacks a program editor which is an important part of any language. BASIC, for instance, allows you to insert new lines as you please and the DELETE command is available for erasing blocks of lines. A program editor is presented in part 3, which deals with compilers, and this could be modified to work with Grafrite. However, there are another few routes open.

Grafrite programs are stored in the computer's memory as pure ASCII text, and nothing manipulates ASCII text better than a wordprocessor. Owners of View, Wordwise / Wordwise Plus or InterWord wordprocessor ROMs have a ready-made program editor, and these can be used to write Grafrite programs, provided that a few rules are followed. In all cases the single command per line must be adhered to; and all programs must have an "@" character at their very end (on a separate line). The following short listing shows this:

```
#A=&200
#B=&1
#X=&0
```

```
#Y=#X
MOVE
SCREEN 5
REPEAT
#X+#B
#Y=#X
DOT
#A-#B
WHILE #A
@
```

The @ character is the last one. This text should then be saved, using SAVE in View and option 1 in Wordwise. It is then necessary to 'format' the text using the Formatter program given in listing 6.1. All this does is to replace the @ sign with an &FF character and re-save the file in the G directory (tape users can amend line 290 to simply re-save the file). The program can then be loaded and used as described. Keep the source file so that you can re-load it and edit it if required.

A final point for View users: do not add a default ruler – use the one supplied when you enter View – and your programs can then be edited in Mode 7 (or Mode 6 for Electron owners.)

Another, albeit more devious, way to edit programs is to use a machine-code monitor program! Simply enter your monitor's ASCII data entry mode and edit the memory contents directly. Remember to place a Return character after each command (ASCII &0D) and a GTOP marker at the end of the program (&FF).

Escape route

When in immediate mode, pressing the ESCAPE key will result in an ⟨Escape⟩ error message. However, this is not the case when a program is in operation – pressing ESCAPE will not have the desired effect and will probably cause the micro to 'hang-up'. This can be resolved very simply. When the ESCAPE key is pressed the MOS sets the top bit (bit 7) of memory location &FF. All we need do is to test this location at a certain point in the program: if the bit is set then the Escape can be effected, otherwise the program runs on uninterrupted. The best time to do this is when the program interpreter is being reset before executing

the next command. The following lines added to listing 4.1 will have the desired effect:

```
3171  BIT &FF
3172  BPL noescape
3173  JMP error
3174  .noescape
```

If you go on to add new, longer, commands then the above segment should be added into the command coding itself as well.

Adding new commands

You will, with luck, want to add new commands to your Grafrite interpreter. As I mentioned at the start of chapter 4, the interpreter has been written in to make the process of expanding it as painless as possible. There are a number of extra graphics commands that can be added fairly easily. The following, for instance, could all be implemented via VDU commands:

COLOUR	– sets the GCOL colour
NORMAL	– equivalent of GCOL 0,x
OR	– equivalent of GCOL 1,x
AND	– equivalent of GCOL 2,x
EOR	– equivalent of GCOL 3,x
INVERT	– equivalent of GCOL 4,x
FOREGROUND	– set foreground colour
BACKGROUND	– set background colour
CLEAR	– clear screen (CLS)
WIPE	– clear graphics screen (CLG)
LOGICAL x,y	– define logical colour
GWINDOW	– define graphics window
ORIGIN	– define the graphics origin
PRINT	– print an ASCII string

The final command in the list cannot be implemented with a VDU command; but, by using a VDU 5 at the start of the Grafrite initialisation process, the text following the PRINT command can be written at the graphics cursor position. Other commands could include CIRCLE, ARC, SHADE and so forth, depending on your programming expertise.

To add an extra command you need to do two things:

- add command to command table
- add machine-code to execute command

Remember that the latter needs to update the 'program' pointer by calling the *endadd* routine; and they must jump to *resetinterp* to re-set the interpreter.

You might also like to add an extra immediate command, HELP, which will list the commands currently implemented. This can be done simply enough, and just needs the commands in the command table to be printed out, skipping the two address bytes as they are encountered. The negative byte of the address can be used as previously described to mark the end of the command name being printed.

Another useful addition would be to allow OS commands to be used in programs. As Grafrite programs are graphic by nature you will want to save screens and perhaps dump them to a printer, to preserve them for posterity! As it stands now, a * command is treated as an immediate command. Add an extra command – perhaps OS – followed by the * command (minus the literal *) and the interpreter can pass the command to OSCLI when encountered. For example:

OSSAVE SCREEN 3000 8000

could pass the line after OS to OSCLI so:

*SAVE SCREEN 3000 8000

could effectively be performed.

Another important addition would be SAVE and LOAD commands to allow you to save and load program files more elegantly. Of course the possibilities are endless – just suit yourself!

Finally, as it stands, the Grafrite interpreter only identifies the characters it needs to. For example, if you enter the command SCREENXZ it will perform the action of SCREEN. Ideally, once the interpreter 'thinks' it has recognised a command it needs to test for any extra letters tagged on to the end of the identified command and, if found, generate a 'Syntax error'.

Tokenising

Grafrite programs are stored in ASCII form – that we know – but, as a consequence, the interpretation process is slowed down. For example, the command SCREEN needs to have six components identified; ie, S, C, R, E, E, and N. A more efficient way of storing a program is to use *tokens*. These were introduced in chapter 4 and were used to identify the end of an immediate command. The same can be done to mark a command in a Grafrite (or any other) program. The idea is that when you enter a command, a *tokenising interpreter* scans a list of keywords and, if it recognises the word, it extracts the appropriate token from another table and puts this into the program, rather than the original ASCII command string. Now, when you come to run the program, the run-time interpreter need only identify a single byte for each command. This has two effects: first, the program runs faster because only a single byte need be identified; and second, the program takes up less memory.

As before, we need a means of telling the run-time interpreter that the byte encountered is a token. To do this we can use our old friend the negative bit, so that all tokens have a value of &80 to &FE. Note, though, that &FF is used to mark the program GTOP; and so, &FF, to avoid possible confusion, should not be used as a token.

The easiest way to implement tokens is simply to assign a single byte to each command: for example, the token for SCREEN might be &80, the token for REPEAT &81, WHILE might be &82 and so on. However, the token has several other unused bits that can convey information to your interpreter when it is expanded to encompass more commands. BASIC programs are tokenised – but the commands and tokens are grouped together as Table 6.1 shows.

Token range	Command type
&80-&84	Operators
&85-&8C	Auxiliary tokens
&8D	line-number token
&8E	BASIC 2's OPENIN
&8F-&93	Pseudo-variable functions
&94-&BC	Numeric functions

&BD-&C4	String functions
&C5	EOF command
&C6-&CD	Main commands
&CE	Not used
&CF-&D3	Pseudo-variable statements
&D4-&FF	Statements

Table 6.1: BBC BASIC's tokens

Each bit of a BASIC token conveys information to the BASIC interpreter; and, by evaluating the bit pattern of the token, the interpreter knows what to do – for example, whether a string should follow the command, if a number should follow it, and so on.

The basic subset of Grafrite commands as they stand now could be grouped together in the manner listed in table 6.2.

Token range	Command type	Commands
&80-&83	Immediate	LIST, NEW, RUN, *
&90–&94	Indirect	DLINE, DOT, LINE MOVE, TRIANGLE
&A0–&A1	Direct	SCREEN, WHILE
&B0	Statement	REPEAT

Table 6.2: Grafrite tokens

The top four bits of each token byte are used to mark the *type* of command, ie, the token &A*x* will inform the interpreter that a direct command is indicated: a suitable subroutine could be called and the number or variable following the command evaluated and stored in a pre-defined buffer. The low four bits identify the *command*, ie, &x4 indicates the command TRIANGLE.

A further extension of the tokenising routine would be a *syntax checker*. In addition to tokenising the command the interpreter would test to make sure that the rest of the line is correct. If it is, it's added to the program; if not, it is rejected with a suitable error message.

Repeating a loop

Use of just a single loop structure within a language is extremely limiting. Grafrite's REPEAT . . . WHILE loop is a useful structure but needs to be extended to allow several loops to be nested within each other. This can be done without too much fuss. All we need do is create a *loop stack* somewhere in memory. Each time a REPEAT is encountered the address it generates is placed on the top of the stack and the loop stack pointer incremented by two. When a WHILE finishes, the stack pointer is decremented by two, to point at any previous REPEAT addresses – this effectively removes the address from the stack. An alternative method would be to use two smaller stacks, one for the high-byte and one for the low-byte of the REPEAT address.

The number of nested loops you can use will depend on the memory available: generally, more than 20 loops should be considered bad practice. Your stack pushing routine should test to see if the stack pointer is equal to the value you assign as the maximum number of embedded loops. If it exceeds this then an error should be generated. Similarly, when an address is pulled from the stack, the stack pointer should be checked again to make sure it will not decrement below 0 as would be the case if an extra WHILE were incorporated into a program!

There are many other types of loops and control structures and in general they are all relatively simple to implement – it is normally the *evaluation* process which creates the problems. The simplest of all loop structures is the GOTO command – in machine-code this performs a readily coded JMP. What is difficult is *marking the destination* of the JMP. In BASIC, line numbers are used, which is effective; but Grafrite does not support line numbers. These could be added if so desired but a better way would be to introduce a *label* system. A label is used to mark a point in a program and the GOTO statement will jump to the address of that label, eg:

```
start
MOVE
#X+#Z
#Y+#C
```

```
LINE
GOTO start
```

The problem here is to get the interpreter to realise that *start* is a variable and not a bad command name! One way would be to prefix the variable with a *marker character* which tells the interpreter 'ignore the rest of this line until you come to a GOTO'. The character '@' could be used so that the short program becomes:

```
@start
MOVE
#X+#Z
#Y+#C
LINE
GOTO start
```

The GOTO execution code could look through memory until it finds an '@' and then *compare* its *target* label with that of the one *just found*. If it is identical it jumps to that address; otherwise, it looks for the next occurrence and so on. A subroutine similar to BASIC's GOSUB . . . RETURN could be added along the same lines. The command ENDPROG could mark the end of the program; beyond this end-part subroutines could be stored, each one assigned a name and called thus:

```
SUBROUTINE screen
SUBROUTINE setmove
SUBROUTINE display
ENDPROG


@screen
SCREEN 5
ALLDONE


@setmove
#X=#A
#Y=#B
MOVE
ALLDONE
```

```
@display
#X=#C
#Y=#D
LINE
ALLDONE
```

Here @ marks the start of a SUBROUTINE destination. The SUBROUTINE code searches above ENDPROG for a '@' and then tries to match the following string. If found, it saves the current address on a suitable stack and executes the code until it encounters an ALLDONE command, at which time the previously saved address is restored and the main program re-entered.

And finally, one plea. Please don't be tempted to use illegal routines within the BASIC ROM – it's cheating! Your language should be completely self-contained, apart from MOS calls, so that you can run it with the BASIC chip removed.

Listing 6.1

```
 1Ø REM Wordwise/View Formatter
 2Ø REM Mastering Interpreters
 3Ø REM and Compilers
 4Ø REM (C) Bruce Smith 1986
 5Ø :
 6Ø DIM os% 4Ø
 7Ø buffer%=&4ØØØ
 8Ø INPUT"Enter filename: "file$
 9Ø PROCoscli("LOAD "+file$+" "+STR$~buffer%)
 91 PRINT "Formatting...."
1ØØ position%=buffer%
11Ø REPEAT
12Ø position%=position%+1
13Ø UNTIL ?position%=ASC("@")
14Ø ?position%=&FF
15Ø position%?1=13
16Ø position%?2=&FF
17Ø position%=position%+3
18Ø PROCsave
19Ø END
2ØØ :
21Ø DEF PROCoscli($os%)
22Ø X%=os%
23Ø Y%=os% DIV 256
```

```
24Ø CALL &FFF7
25Ø ENDPROC
26Ø :
27Ø DEF PROCsave
271 PRINT "Saving file as: G.";file$
28Ø position%=position%+1
29Ø W$="SAVE "+"G."+file$+" "+STR$~buffer%+"
    "+STR$~position%
3ØØ PROCoscli(W$)
31Ø ENDPROC
```

THREADED INTERPRETIVE LANGUAGES

ABOUT TILs

The phrase 'Threaded Interpretive Language' (TIL) implies an interpreted language. However, this is not strictly true – a TIL is both interpreted and compiled. It does not compile machine-code in the sense of a true compiler but instead compiles the addresses of addresses of machine code. That probably sounds like double-Dutch but is quite understandable, as we'll see later on. First, though, let's examine the implementation of a TIL-type language.

A typical TIL consists of a dictionary with entries. Each entry is composed of two parts, the *header* and the *body*.

The header contains, among other information, the name of the command which will invoke its action, just like the keyword used to form the command table of an interpreter such as Grafrite (in the last section of this book). The *keyword* name is followed by two addresses. The second of these addresses is the *command execution address*, ie, the address jumped to to begin execution of the command. The first address, stored directly after the keyword, is the *link address* and points to the start of the previous dictionary entry. In this way the commands are linked or threaded together.

As you may have guessed, the body of the dictionary entry contains the machine-code which performs the

actions required by the command – though this is not always so, as we shall see in the next chapter.

Figure 7.1 shows diagrammatically the construction of a dictionary entry. Let's first look at the *head* in more detail. Several formats can be used – I'll describe perhaps the most common one and no doubt you'll come up with your own later. The head itself can be broken down into three parts.

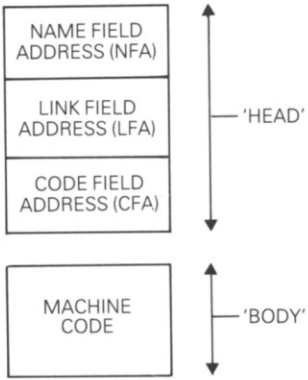


Figure 7.1 A typical TIL entry.

First there is the *Name Field Address* or NFA. In fact this is not an address and as such its title seems slightly misleading – it contains the name of the keyword, preceded by a single byte which conveys the length of the keyword. Thus if the name of the keyword were BEEP the NFA would contain 4.

The NFA is followed by the *Link Field Address* (LFA) and this contains the address of the NFA of the previous keyword. Notice that, since the first entry in the TIL dictionary cannot point to any previous entry, its final link field address will contain 0.

The final head entry is the *Code Field Address* (CFA) and this contains the address of the machine code to be executed.

Figure 7.2 shows how three dictionary entries, called BEEP, CLS and HOME, are linked together.

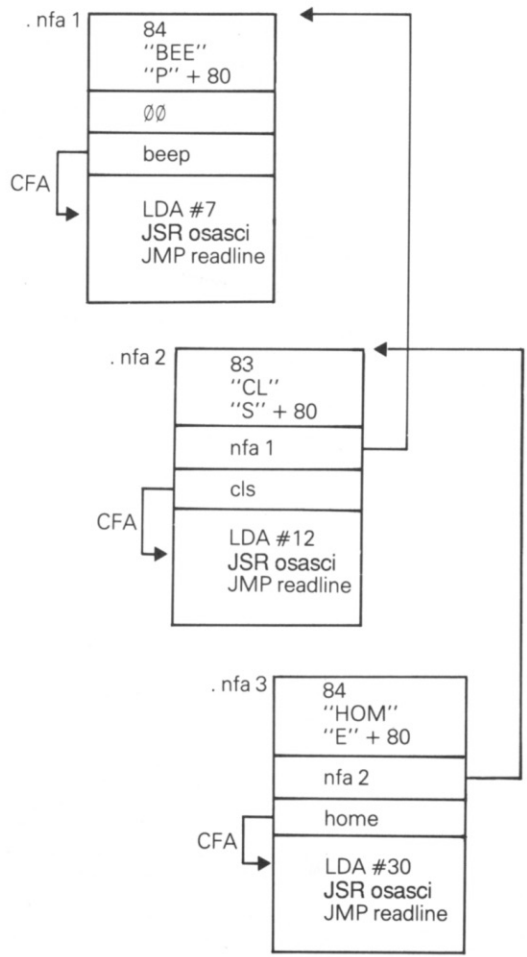


Figure 7.2 How dictionary entries are linked.

The assembly listings below show how these would be assembled into a program:

```
.nfa1   OPT  FNequb (&80+4)          \ length +
                                       negative
                                       bit
                                      (so add &80)
        OPT FNequs ("BEE")           \ command
                                       name, less
                                       last letter
        OPT FNequb ("&80+ASC("P")) \ last letter
                                       plus  neg.
                                       bit
                                       (so add &80)
        OPT  FNequw  (0)              \ no link
                                       address as
                                       first entry
        OPT FNequw (beep)            \ code field
                                       address of
                                       BEEP
.beep   LDA #7
        JSR osasci                   \ do BEEP
        JMP readline                 \ get next
                                       command


.nfa2   OPT FNequb (&80+3)           \ length +
                                       negative bit
        OPT FNequs ("CL")            \ command
                                       name, less
                                       last
                                       letter
        OPT FNequb ("&80+ASC("S")) \ last letter
                                       plus neg.
                                       bit
        OPT FNequw (nfa1)            \ link address
                                       to BEEP NFA
        OPT FNequw (cls)             \ code field
                                       address of
                                       CLS
.cls    LDA #12
        JSR osasci                   \ do CLS
        JMP readline                 \ get next
                                       command
```

```
.nfa3  OPT FNequb (&80+4)          \ length +
                                     negative
                                     bit
       OPT FNequs ("HOM")          \ command
                                     name, less
                                     last letter
       OPT FNequb ("&80+ASC("E")) \ last letter
                                     plus neg.
                                     bit
       OPT FNequw (nfa2)           \ link
                                     address to
                                     CLS NFA
       OPT FNequw (home)           \ code field
                                     address of
                                     HOME
.home  LDA #30
       JSR osasci                  \ do HOME
       JMP readline                \ get next
                                     command
```

The listing throws up a few points which need explaining. In it &80 is added to the NFA length byte, and also to the last character in the keyword, for labelling purposes to mark the start and end of the keyword; in Grafrite &80 was added to the execution addresses in the Grafrite command table for the same reason. The bodies of the dictionary entries all end with the instruction:

JMP readline

This simply returns control to the keyboard scanning routine which itself can be implemented with OSWORD 0.

Searching the dictionary

When a command is typed at the keyboard, the TIL's dictionary must be searched to see if the command can be identified. If it can then the code which forms its body can be executed, otherwise the link address must be extracted from the LFA and the search routine started again at the NFA to which this link address points. This sounds a little complicated, so let's go through it step-by-step before looking at the assembly language which will perform the search.

Search:

1) Get address of last NFA in dictionary
2) Is command same as this one?
3) If YES then extract CFA and jump to this
4) If NO then get LFA
5) If LFA = 0 then command not found; produce an error
6) Otherwise go to step 2.

The following assembly code segment does all the above and uses the following variables:

topaddr : pointer to NFA of last dictionary entry entry
pointer : vectored address into NFA
lbuffer : vectored address into command line buffer
address : jump address buffer

```
.search
LDA #topaddr MOD 256
STA pointer
STA address
LDA #topaddr DIV 256
STA pointer+1
STA address+1              \ point vectors to
                             last NFA

.outer
LDY #0                     \ initialise indirect
                             index

.inner
INY                        \ point to indexed
                             letter of keyword
LDA (pointer),Y            \ get byte from NFA
EOR (lbuffer),Y            \ Exclusive OR with
                             byte in buffer
BEQ inner                  \ if 0 then branch
BPL linkaddress            \ if positive then
                             move to next NFA
INY:INY:INY                \ point Y to CFA
TYA:CLC                    \ move Y into A and
                             get ready to add
ADC address                \ add to address
                             low-byte
```

```
STA address              \ and save
BCC over                 \ branch if no carry
INC address+1            \ else bump high byte
.over
JMP address-1            \ jump to jump buffer
```

The *search* routine does not use a straight CMP instruction, but illustrates another method of testing to see if two bytes are equal. The EOR instruction will leave a bit in the position being tested only if the two bits being tested are unlike; if they are alike, a zero will be returned. The EOR rules are:

0 EOR 0 = 0
1 EOR 0 = 1
0 EOR 1 = 1
1 EOR 1 = 0

The ASCII code for "C" is &43, which in binary is 0100 0011. If the line buffer also contains ASCII "C" the next result of the EOR will be zero, because:

&43	0100 0011
&43	0100 0011
EOR	0000 0000

On the other hand, if the line buffer contained "E" (&45) the result will be non-zero because the result is &06:

&43	0100 0011
&45	0100 0101
EOR	0000 0110

If the first non-zero byte is negative the command has been correctly identified and the BPL *linkaddress* branch will not take place. At this time the Y register is pointing to the last byte in the keyword and must be incremented by three to move it on to point to the first byte in the CFA. The address given by (pointer),Y is placed into *address*. A jump to (*address*−*1*) is finally performed. We know that *address* and *address*+*1* contain the address of the CFA of the identified command. The byte at *address*−*1* must contain the byte &6C which is the opcode for an indirect jump. So, by performing the operation:

JMP address−1

we will be jumping to an instruction that will perform

JMP (address)

That is, jumping to the address held inside the location that *address* is pointing to! As *address* contains the CFA of the identified command, this will jump to the code forming the *body* of the identified command.

Address	Contents	Comments
5000	&84	Marker/length byte
5001	"BEE"	Three ASCII characters
5004	"P"+&80	End marker
5005	&0000	No LFA address
5007	&5009	Code Field address
5009	LDA #7	Bell Character
500B	JSR osasci	Print it
500E	JMP readline	Next command

Figure 7.3

Figure 7.3 tries to show exactly what this all means. It assumes that the command BEEP has been identified and is stored in computer memory from &5000. The CFA address is &5007 and it is this address that has been extracted and placed at *address*. So the JMP (*address*) instruction will jump to the address held at &5007. This is &5009, which is where the code for BEEP begins. This may seem a rather roundabout way to cause the execution of a bit of code, but there is a very good reason for doing it this way, as we shall see.

Listing 7.1 implements what I have called a mini-TIL. It is not a language in the true sense in that it does not re-direct BRKV, trap the BREAK key or reset the stack as discussed previously. It illustrates how a very simple TIL can be constructed. It uses the three commands discussed so far, plus one more: COMMANDS. This is a type of help command that prints out all the commands available in the MiniTIL dictionary. It does this by using the LFA addresses to step from one NFA to the next, printing each keyword as it is encountered.

Entering MiniTIL

As with all the programs in this book, most lines are limited to one instruction per line, and only doubled up when the instructions are short. The listing includes a checksum routine which will sum the code when it has been assembled from &5000. The assembly address should only be changed after the program has been run, to ensure that it is indeed correct. If the checksum is incorrect this will be reported.

Once the source code is assembled correctly, save it using a suitable filename, ie "MINITIL". The machine-code itself could be saved with:

*SAVE TIL 5000 5160

and subsequently re-loaded and run with,

*RUN TIL (or simply *TIL for non-tape based systems).

A function, *FNtoken*, has been added which simplifies the coding of the first part (the NFA) of the dictionary header. This should be called with the command name passed as a string parameter. *FNtoken* will then create the NFA of the header.

The main areas of the program, some of which have been described already, are detailed in table 7.1 below.

Routine	lines	Description
thorf	290 to 360	Select Mode 7 screen and put &6C at *address−1*
setloop	370 to 490	Print start-up screen details
readline	510 to 770	Print okay prompt: use OSWORD 0 to read line: jump to *escape* if ESCAPE pressed
return	790 to 810	Test to see if input buffer empty
search	830 to 890	Point to last NFA
outer	910 to 1060	Locate command, extract CFA and jump there. Otherwise abort
linkaddress	1080 to 1230	Locate next LFA and place it in the vector called *pointer*
error	1250	entry for unrecognised command
eloop	1270 to 1340	Error print loop
message	1360 to 1380	Error message store
escape	1400 to 1440	Escape detection routine

nfa1	1460 to 1810	Dictionary entry for COMMANDS
nfa2	1830 to 1900	Dictionary entry for BEEP
nfa3	1920 to 1990	Dictionary entry for CLS
topaddr	2010 to 2080	Dictionary entry for HOME

Table 7.1: The main routines of listing 7.1.

Adding new dictionary entries

Extra entries can be added to the MiniTIL dictionary fairly easily. All that is needed is to enter the code at the correct point and adjust any Link Field Addresses. The best place to add a new entry is directly before *topaddr*, which starts on line 2010. A new entry could have a NFA label of *nfa4*, and this should be pointed to by editing the LFA of the HOME dictionary entry in line 2030 accordingly. Remember that the code body of a keyword must *always* end with a jump to *readline*. For example, to add the command SIXMODE (change screen to mode 6) to the dictionary you could enter it into the main listing thus:

```
1991 .nfa4
1992 OPT FNtoken ("SIXMODE")
1993 OPT FNequw  (nfa3)
1994 OPT FNequw  (sixmode)
1995 .sixmode
1996 LDA #22
1997 JSR oswrch
1998 LDA #6
1999 JSR oswrch
2000 JMP readline
```

The LFA of HOME must be pointed to *nfa4* by editing line 2030:

2030 OPT FNequw (nfa4)

Re-running the program will now produce a checksum error because of the extra code, so delete line 80. The command SIXMODE can now be executed and it will appear in the list of Commands.

A close encounter of the Forth kind

I said at the beginning of this chapter that a true TIL will compile sets of addresses which point to machine code. Well, as you may have already gathered, this is

not strictly true: it compiles the addresses of memory locations which contain the addresses of machine code. If you followed the reasoning behind the last few pages this should now be easy to understand. The compiled addresses are those of each command's Code Field Address (CFA). A typical TIL implemented along these lines is the language FORTH. A description of FORTH is given in Chapter 8, but for now I'll confine myself to showing how lists of addresses can be compiled and executed to result in my TIL, aptly called miniTHORF.

Dictionary entries in miniTHORF consist of a head and body – no different from that described earlier – but now the body of a dictionary entry does not consist of machine code. Instead it consists of a *list of addresses*: to be exact, a list of CFAs of the machine code to be executed. What we clearly need now is a *control mechanism* which will step through the body of the dictionary entry extracting each CFA in order and causing its execution. To do this we need to keep two zero-page vectors.

The first of these is the *Interpretive Pointer* (IP) which is used to hold the address of the *next* CFA address to be extracted from the body of the dictionary entry.

The second is the *Code Field Pointer* itself (W) and this contains the address of the CFA of the machine code *currently* being executed.

The machine-code mechanism to step through the body of a dictionary entry is called *next* and is listed below:

```
.next
LDY #1          \ set indirect index
LDA (IP),Y      \ get high byte CFA
STA W+1         \ save in jump buffer
DEY
LDA (IP),Y      \ get low byte CFA
STA W           \ save in jump buffer
CLC
LDA IP          \ get low byte of pointer
ADC #2          \ add 2 to go to next CFA
STA IP          \ save result
BCC nocarry     \ branch if carry clear
```

```
INC IP+1          \ else bump high bytes
.nocarry
JMP W-1           \ jump to indirect jump
```

This routine must be jumped to by each command to make sure that each CFA is extracted in turn. This is easy enough to do and just needs each command execution code to end in a JMP *next* instruction.

The other mechanisms need to be set up. First, we need to set the *next* routine into operation, which can be done by jumping to it after a line is read in from the keyboard. We also need a way of allowing the routine to *exit* back to the keyboard scanning routine. This can be done in a similar fashion by including the CFA address of a routine called *exit* at the base of the body of the dictionary entry. The use of W as a variable name is because this is often referred to as the Word Address Register.

The assembly code for CLEAR shows how a command called "CLEAR" could be defined – this first clears the screen and then performs a beep by calling two dictionary entries in turn – CLS and BEEP. These were defined earlier and only differ in that each now ends with a jump to *next*.

```
.topaddr
 OPT FNequb (&85)          \ first NFA byte
 OPT FNequs ("CLEA")       \ keyword string
 OPT FNequb (ASC"R"+&80)   \ terminating byte
 OPT FNequw (add4)         \ LFA  to previous NFA
 OPT FNequw (cfacls)       \ CLS cfa address
 OPT FNequw (cfabeep)      \ BEEP cfa address
 OPT FNequw (cfaexit)      \ CFA of exit routine
```

To see how the Interpretive Pointer and Code Field Address pointers work, study figure 7.4, which illustrates the situation when CLEAR is being executed.

Let us suppose that we are half-way through the definition and that the dictionary entry for CLS is currently being executed. At this stage, *next* has already extracted the cfa of CLS and put it into W. Now W, the code field pointer, contains the *code field*

address of CLS. Also, *next* has acted so that the interpretive pointer, IP, now contains the address of the next address to be executed – the CFA of BEEP, in fact.

When the execution of CLS has finished, control will be returned to *next*. This will move the contents of the IP across into W, which will now contain the next address to be indirectly jumped to.

The contents of IP are then incremented by two so that it then contains the next CFA to be executed.

The final CFA in the dictionary entry for CLEAR is that of *exit* which will return control back to the keyboard.

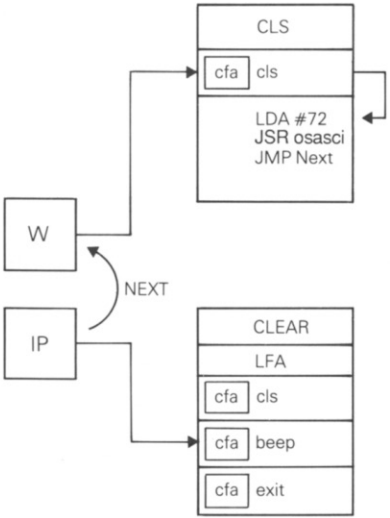


Figure 7.4 Flow of a FORTH word being executed.

During action of CLEAR, W holds the CFA of the word being executed. IP holds the CFA of the next word to be executed. When cls has completed, contents of IP will be passed into W and IP will be incremented by 2, to point to next word address.

The *exit* routine is really quite simple and is listed below:

```
.exit
OPT FNequw (exexit)          \ CFA of (EXIT)
.exexit
JMP next                     \ goto next
```

As you can see, this section of code is like a cut-down dictionary entry, having just a minimal head and a body – no NFA, no LFA. For this reason, sections of code such as these are often referred to as *headerless* entries. Brackets are used to distinguish headerless entries from normal ones, ie (EXIT) is a headerless entry and EXIT a normal entry.

Going THORF

Listing 7.2 contains a minimal amount of assembly code to provide you with a small THORF TIL along the lines described. As all commands now consist only of addresses, a base of machine-code routines must be provided for them to act on. These machine-code routines are termed *primitives*. The dictionary entries which call them are termed *secondaries*.

When our little THORF is run there are six commands available, and COMMANDS will list them. Each command is a *secondary* which calls its *primitive level*. The larger this base level is made, the more secondary level commands can be concocted – so much so that large programs can be put into operation just by a single command. A THORF language would be particularly suited to control purposes.

As with miniTIL, miniTHORF assembles code from &5000 and this can be altered to suit your own needs by changing the value assigned to P%. However, this should be done only after the program is running correctly, as a checksum routine is included in the listing. The machine code of miniTHORF is under 0.5k in length and can be saved using,

```
*SAVE THORF 5000 +200
```

Obviously, these figures should be changed if you reassemble the machine code elsewhere. Use a similar filename to save the source listing before attempting to run it, ie, STHORF.

Table 7.2 lists the main routines within the miniTHORF listing, most of which have been detailed elsewhere in this chapter.

Routine	Lines	Description
throf	240 to 310	Set mode and set indirect opcode
setloop	320 to 440	Print start-up message
readline	460 to 720	Print prompt: readline test for ESCAPE
return	740 to 760	Redo if buffer empty
search	780 to 840	Set top of dictionary
outer	860 to 1010	Search dictionary for command
linkaddr	1030 to 1180	Link to next entry if not identified
error	1200 to 1210	Entry for unknown command
eloop	1220 to 1290	Print error message loop
message	1310 to 1330	Error message buffer
escape	1350 to 1390	Acknowledge ESCAPE
next	1410 to 1550	Execute next CFA
exit	1570 to 1660	Leave *next* and go to keyboard
cfacommand	1630 to 1950	Primary for COMMANDS
cfabeep	1980 to 2030	Primary for BEEP
cfacls	2060 to 2110	Primary for CLS
cfahome	2130 to 2190	Primary for HOME
cfacr	2220 to 2260	Primary for RETURN
nfa1	2280 to 2310	Secondary for COMMANDS
nfa2	2330 to 2360	Secondary for CLS
nfa3	2380 to 2410	Secondary for HOME
nfa4	2430 to 2460	Secondary for BEEP
nfa5	2480 to 2510	Secondary for RETURN
topaddr	2530 to 2610	Secondary for DOALL

Table 7.2: The main routines in miniTHORF

Extending miniTHORF

As with MiniTIL, miniTHORF can be extended simply by adding extra dictionary entries at the top of the dictionary, making sure that each entry is linked to the previous one and that the very last entry is marked as *topaddr*. The language at present is a 'programless' in that it cannot be used to develop a THORF program, but this feature could be incorporated if so desired; more complex commands would obviously require data to work on, so general routines for handling these need to be added. These structures are somewhat familiar to FORTH itself, which brings us nicely to Chapter 8!

Listing 7.1

```
10 REM MiniTIL
20 REM Mastering Interprters
30 REM and Compilers
40 REM (C) Bruce Smith 1986
60 PROCvariables
70 PROCassemble
80 PROCchecksum
90 CALL throf
100 END
110 :
120 DEF PROCvariables
130 address=&71
140 pointer=&75
150 store=&7E
160 param=&80
170 linebuffer=&C01
180 lbuffer=&86
190 lb=linebuffer-1
200 oswrch=&FFEE
210 osword=&FFF1
220 osbyte=&FFF4
230 ENDPROC
240 :
250 DEF PROCassemble
260 FOR pass=0 TO 3 STEP 3
270 P%=&5000
280 [OPT pass
290 .throf
300 LDA #&6C
310 STA address-1
320 LDA #22
330 JSR oswrch
340 LDA #7
350 JSR oswrch
360 LDX #&FF
370 .setloop
380 INX
390 LDA header,X
400 BEQ readline
410 JSR &FFE3
420 BNE setloop
430 .header
440 OPT FNequw (&0D0C)
```

```
45Ø OPT FNequs ("Acorn MOS")
46Ø OPT FNequw (&ØDØD)
47Ø OPT FNequs ("MiniTIL")
48Ø OPT FNequw (&ØDØD)
49Ø OPT FNequb (Ø)
5ØØ \
51Ø .readline
52Ø LDA #ASC("O")
53Ø JSR &FFE3
54Ø LDA #ASC("K")
55Ø JSR &FFE3
56Ø LDA #32
57Ø JSR &FFE3
58Ø LDA #lb MOD 256
59Ø STA lbuffer
6ØØ LDA #lb DIV 256
61Ø STA lbuffer+1
62Ø LDA #linebuffer MOD 256
63Ø STA param
64Ø LDA #linebuffer DIV 256
65Ø STA param+1
66Ø LDA #128
67Ø STA param+2
68Ø LDA #32
69Ø STA param+3
7ØØ LDA #255
71Ø STA param+4
72Ø LDX #param MOD 256
73Ø LDY #param DIV 256
74Ø LDA #Ø
75Ø JSR osword
76Ø BCC return
77Ø JMP escape
78Ø \
79Ø .return
8ØØ CPY #Ø
81Ø BEQ readline
82Ø \
83Ø .search
84Ø LDA #topaddr MOD 256
85Ø STA pointer
86Ø STA address
87Ø LDA #topaddr DIV 256
```

```
88Ø STA pointer+1
89Ø STA address+1
9ØØ \
91Ø .outer
92Ø LDY #Ø
93Ø .inner
94Ø INY
95Ø LDA (pointer),Y
96Ø EOR (lbuffer),Y
97Ø BEQ inner
98Ø BPL linkaddress
99Ø INY:INY:INY
1ØØØ TYA:CLC
1Ø1Ø ADC address
1Ø2Ø STA address
1Ø3Ø BCC over
1Ø4Ø INC address
1Ø5Ø .over
1Ø6Ø JMP address-1
1Ø7Ø \
1Ø8Ø .linkaddress
1Ø9Ø INY
11ØØ LDA (pointer),Y
111Ø BPL linkaddress
112Ø INY
113Ø LDA (pointer),Y
114Ø STA store
115Ø INY
116Ø LDA (pointer),Y
117Ø BEQ error
118Ø STA pointer+1
119Ø STA address+1
12ØØ LDA store
121Ø STA pointer
122Ø STA address
123Ø JMP  outer
124Ø \
125Ø .error
126Ø LDX #255
127Ø .eloop
128Ø INX
129Ø LDA message,X
13ØØ BNE not
```

```
131Ø JMP readline
132Ø .not
133Ø JSR &FFE3
134Ø BNE eloop
135Ø \
136Ø .message
137Ø OPT FNmessage ("?command")
138Ø OPT FNmessage ("Escape")
139Ø \
14ØØ .escape
141Ø LDA #126
142Ø JSR osbyte
143Ø LDX #12
144Ø BNE eloop
145Ø \
146Ø .add1
147Ø OPT FNtoken ("COMMANDS")
148Ø OPT FNequw (Ø)
149Ø OPT FNequw (commands)
15ØØ .commands
151Ø LDA #topaddr MOD 256
152Ø STA store
153Ø LDA #topaddr DIV 256
154Ø STA store+1
155Ø .vloop
156Ø LDY #Ø
157Ø .pvloop
158Ø INY
159Ø LDA (store),Y
16ØØ BMI donethis
161Ø JSR oswrch
162Ø BNE pvloop
163Ø .donethis
164Ø AND #&7F
165Ø JSR oswrch
166Ø LDA #&2Ø
167Ø JSR oswrch
168Ø INY
169Ø LDA (store),Y
17ØØ PHA
171Ø INY
172Ø LDA (store),Y
173Ø BEQ doneall
```

```
174Ø STA store+1
175Ø PLA
176Ø STA store
177Ø JMP vloop
178Ø \
179Ø .doneall
18ØØ PLA
181Ø JMP readline
182Ø \
183Ø .add2
184Ø OPT FNtoken ("BEEP")
185Ø OPT FNequw (add1)
186Ø OPT FNequw (beep)
187Ø .beep
188Ø LDA #7
189Ø JSR &FFE3
19ØØ JMP readline
191Ø \
192Ø .add3
193Ø OPT FNtoken ("CLS")
194Ø OPT FNequw (add2)
195Ø OPT FNequw (cls)
196Ø .cls
197Ø LDA #12
198Ø JSR &FFE3
199Ø JMP readline
2ØØØ \
2Ø1Ø .topaddr
2Ø2Ø OPT FNtoken ("HOME")
2Ø3Ø OPT FNequw (add3)
2Ø4Ø OPT FNequw (home)
2Ø5Ø .home
2Ø6Ø LDA #3Ø
2Ø7Ø JSR &FFE3
2Ø8Ø JMP readline
2Ø9Ø \
21ØØ ]
211Ø NEXT
212Ø ENDPROC
213Ø :
214Ø DEF FNtoken (A$)
215Ø L%=LEN(A$)+&8Ø
216Ø [OPT pass
```

```
217Ø OPT FNequb (L%)
218Ø OPT FNequs (A$)
219Ø ]
22ØØ P%=P%-1
221Ø M%=?P%+&8Ø
222Ø [OPT pass
223Ø OPT FNequb (M%)
224Ø ]
225Ø =pass
226Ø :
227Ø DEF FNequb (B%)
228Ø ?P%=B%
229Ø P%=P%+1
23ØØ =pass
231Ø :
232Ø DEF FNequw (W%)
233Ø ?P%=W% MOD 256
234Ø P%?1=W% DIV 256
235Ø P%=P%+2
236Ø =pass
237Ø :
238Ø DEF FNequs (string$)
239Ø $P%=string$
24ØØ P%=P%+LEN($P%)
241Ø =pass
242Ø :
243Ø DEF FNmessage (mess$)
244Ø [
245Ø OPT FNequw (&ØDØ7)
246Ø OPT FNequs (mess$)
247Ø OPT FNequw (&ØØØD)
248Ø ]
249Ø =pass
25ØØ :
251Ø DEF PROCchecksum
252Ø X%=Ø
253Ø FOR L%=&5ØØØ TO &5155
254Ø X%=X%+?L%
255Ø NEXT
256Ø IF X%=4Ø798 THEN ENDPROC
257Ø VDU 7
258Ø PRINT'"Checksum error!"
259Ø STOP
```

Listing 7.2

```
10 REM miniTHORF
20 REM Mastering Interprters
30 REM and Compilers
40 REM (C) Bruce Smith 1986
50 :
60 PROCvariables
70 PROCassemble
80 PROCchecksum
90 CALL thorf
100 END
110 :
120 DEF PROCvariables
130 W=&7A:IP=&7C:param=&80
140 lbuffer=&8A:store=&8C
150 lb=&C00:linebuffer=&C01
160 oswrch=&FFEE:osasci=&FFE3
170 osword=&FFF1:osbyte=&FFF4
180 ENDPROC
190 :
200 DEF PROCassemble
210 FOR pass=0 TO 3 STEP 3
220 P%=&5000
230 [OPT pass
240 .thorf
250 LDA #&6C
260 STA W-1
270 LDA #22
280 JSR oswrch
290 LDA #7
300 JSR oswrch
310 LDX #&FF
320 .setloop
330 INX
340 LDA header,X
350 BEQ readline
360 JSR osasci
370 BNE setloop
380 .header
390 OPT FNequw (&0D0C)
400 OPT FNequs ("Acorn MOS")
410 OPT FNequw (&0D0D)
420 OPT FNequs ("miniTHORF")
430 OPT FNequw (&0D0D)
```

```
44Ø OPT FNequb (Ø)
45Ø \
46Ø .readline
47Ø LDA #ASC("O")
48Ø JSR osasci
49Ø LDA #ASC("K")
5ØØ JSR osasci
51Ø LDA #32
52Ø JSR osasci
53Ø LDA #lb MOD 256
54Ø STA lbuffer
55Ø LDA #lb DIV 256
56Ø STA lbuffer+1
57Ø LDA #linebuffer MOD 256
58Ø STA param
59Ø LDA #linebuffer DIV 256
6ØØ STA param+1
61Ø LDA #128
62Ø STA param+2
63Ø LDA #32
64Ø STA param+3
65Ø LDA #255
66Ø STA param+4
67Ø LDX #param MOD 256
68Ø LDY #param DIV 256
69Ø LDA #Ø
7ØØ JSR osword
71Ø BCC return
72Ø JMP escape
73Ø \
74Ø .return
75Ø CPY #Ø
76Ø BEQ readline
77Ø \
78Ø .search
79Ø LDA #topaddr MOD 256
8ØØ STA IP
81Ø STA W
82Ø LDA #topaddr DIV 256
83Ø STA IP+1
84Ø STA W+1
85Ø \
86Ø .outer
```

```
87Ø LDY #Ø
88Ø .inner
89Ø INY
9ØØ LDA (IP),Y
91Ø EOR (lbuffer),Y
92Ø BEQ inner
93Ø BPL linkaddress
94Ø INY:INY:INY
95Ø TYA:CLC
96Ø ADC IP
97Ø STA IP
98Ø BCC over
99Ø INC IP+1
1ØØØ .over
1Ø1Ø JMP next
1Ø2Ø \
1Ø3Ø .linkaddress
1Ø4Ø INY
1Ø5Ø LDA (IP),Y
1Ø6Ø BPL linkaddress
1Ø7Ø INY
1Ø8Ø LDA (IP),Y
1Ø9Ø STA store
11ØØ INY
111Ø LDA (IP),Y
112Ø BEQ error
113Ø STA IP+1
114Ø STA W+1
115Ø LDA store
116Ø STA IP
117Ø STA W
118Ø JMP  outer
119Ø \
12ØØ .error
121Ø LDX #255
122Ø .eloop
123Ø INX
124Ø LDA message,X
125Ø BNE not
126Ø JMP readline
127Ø .not
128Ø JSR osasci
129Ø BNE eloop
```

```
13ØØ \
131Ø .message
132Ø OPT FNmessage ("?command")
133Ø OPT FNmessage ("Escape")
134Ø \
135Ø .escape
136Ø LDA #126
137Ø JSR osbyte
138Ø LDX #12
139Ø BNE eloop
14ØØ \
141Ø .next
142Ø LDY #1
143Ø LDA (IP),Y
144Ø STA W+1
145Ø DEY
146Ø LDA (IP),Y
147Ø STA W
148Ø CLC
149Ø LDA IP
15ØØ ADC #2
151Ø STA IP
152Ø BCC nocarry
153Ø INC IP+1
154Ø .nocarry
155Ø JMP W-1
156Ø \
157Ø .exit
158Ø OPT FNequw (exexit)
159Ø .exexit
16ØØ JMP readline
161Ø \
162Ø \Code for COMMANDS
163Ø .cfacommands
164Ø OPT FNequw (commands)
165Ø .commands
166Ø LDA #topaddr MOD 256
167Ø STA store
168Ø LDA #topaddr DIV 256
169Ø STA store+1
17ØØ .vloop
171Ø LDY #Ø
172Ø .pvloop
```

```
173Ø INY
174Ø LDA (store),Y
175Ø BMI donethis
176Ø JSR oswrch
177Ø BNE pvloop
178Ø .donethis
179Ø AND #&7F
18ØØ JSR oswrch
181Ø LDA #&2Ø
182Ø JSR oswrch
183Ø INY
184Ø LDA (store),Y
185Ø PHA
186Ø INY
187Ø LDA (store),Y
188Ø BEQ doneall
189Ø STA store+1
19ØØ PLA
191Ø STA store
192Ø JMP vloop
193Ø .doneall
194Ø PLA
195Ø JMP next
196Ø \
197Ø \ Code for BEEP
198Ø .cfabeep
199Ø OPT FNequw (beep)
2ØØØ .beep
2Ø1Ø LDA #7
2Ø2Ø JSR osasci
2Ø3Ø JMP next
2Ø4Ø \
2Ø5Ø \ Code for CLS
2Ø6Ø .cfacls
2Ø7Ø OPT FNequw (cls)
2Ø8Ø .cls
2Ø9Ø LDA #12
21ØØ JSR osasci
211Ø JMP next
212Ø \
213Ø \ Code for HOME
214Ø .cfahome
215Ø OPT FNequw (home)
```

```
216Ø .home
217Ø LDA #3Ø
218Ø JSR osasci
219Ø JMP next
22ØØ \
221Ø \ Code for RETURN
222Ø .cfacr
223Ø OPT FNequw (creturn)
224Ø .creturn
225Ø JSR &FFE7
226Ø JMP next
227Ø \
228Ø .nfa1
229Ø OPT FNhead ("COMMANDS",Ø)
23ØØ OPT FNequw  (cfacommands)
231Ø OPT FNequw  (exit)
232Ø \
233Ø .nfa2
234Ø OPT FNhead ("CLS",nfa1)
235Ø OPT FNequw (cfacls)
236Ø OPT FNequw (exit)
237Ø \
238Ø .nfa3
239Ø OPT FNhead ("HOME",nfa2)
24ØØ OPT FNequw (cfahome)
241Ø OPT FNequw (exit)
242Ø \
243Ø .nfa4
244Ø OPT FNhead ("BEEP",nfa3)
245Ø OPT FNequw (cfabeep)
246Ø OPT FNequw (exit)
247Ø \
248Ø .nfa5
249Ø OPT FNhead ("RETURN",nfa4)
25ØØ OPT FNequw (cfacr)
251Ø OPT FNequw (exit)
252Ø \
253Ø .topaddr
254Ø OPT FNhead ("DOALL",nfa5)
255Ø OPT FNequw (cfacls)
256Ø OPT FNequw (cfabeep)
257Ø OPT FNequw (cfacr)
258Ø OPT FNequw (cfacr)
```

```
259Ø OPT FNequw (cfacommands)
26ØØ OPT FNequw (cfahome)
261Ø OPT FNequw (exit)
262Ø ]
263Ø NEXT
264Ø ENDPROC
265Ø :
266Ø DEF FNhead (A$,addr)
267Ø L%=LEN(A$)+&8Ø
268Ø [OPT pass
269Ø OPT FNequb (L%)
27ØØ OPT FNequs (A$)
271Ø ]
272Ø P%=P%-1
273Ø M%=?P%+&8Ø
274Ø [OPT pass
275Ø OPT FNequb (M%)
276Ø OPT FNequw (addr)
277Ø ]
278Ø =pass
279Ø :
28ØØ DEF FNequb (B%)
281Ø ?P%=B%
282Ø P%=P%+1
283Ø =pass
284Ø :
285Ø DEF FNequw (W%)
286Ø ?P%=W% MOD 256
287Ø P%?1=W% DIV 256
288Ø P%=P%+2
289Ø =pass
29ØØ :
291Ø DEF FNequs (string$)
292Ø $P%=string$
293Ø P%=P%+LEN($P%)
294Ø =pass
295Ø :
296Ø DEF FNmessage (mess$)
297Ø [
298Ø OPT FNequw (&ØDØ7)
299Ø OPT FNequs (mess$)
3ØØØ OPT FNequw (&ØØØD)
3Ø1Ø ]
```

```
302Ø =pass
303Ø :
304Ø DEF PROCchecksum
305Ø X%=Ø
306Ø FOR L%=&5ØØØ TO &51AA
307Ø X%=X%+?L%
308Ø NEXT
309Ø IF X%=512Ø5 THEN ENDPROC
31ØØ VDU 7
311Ø PRINT'"Checksum error!"
312Ø STOP
```

FIRST, SECOND, THIRD . . .

FORTH is the classic example of a Threaded Interpretive Language. Invented in 1969 by Charles Moore, it was designed as a convenient means of controlling equipment.

There are several advantages in using a TIL such as FORTH. Generally, it is faster than the average pure interpreted language, anywhere between 3 and 10 times depending on the application. Number-crunching in FORTH is almost as efficient as its machine code equivalent; memory requirements to implement a TIL are much less (a basic core system would normally run out at around 2-3k in length compared to a 16k equivalent BASIC interpreter); extra commands are created simply by loading in new dictionary entries: this can be compared to, perhaps, installing a new ROM chip to enhance BASIC. Possibly its greatest asset, though, is its extensibility: you can add new commands, usually called *words*, to its dictionary simply by defining them using words already existing in the dictionary. As new words are created these can in turn be used to define even more new words. What's more, a just-defined word can be tested immediately, simply by typing it in at the keyboard!

A FORTH interpreter would be similar to the combined actions of the two listings shown in the last chapter, whereby both primitive and secondary dictionary entries can be executed and/or included in new dictionary entries. Space does not allow a minimal FORTH system to be listed here – such a program would be about three times as long as the Grafrite listing! However, a general discussion of FORTH, with sample routines, follows. Armed with this, and a copy of a FIG FORTH listing, implementing the system or

creating your own should present no real problems. FIG, by the way, stands for FORTH Interest Group, which is very active in the UK and details of which can be found in Appendix C – suffice it to say that a FORTH listing from them (for the 6502) will cost less than the price of this book!

Stacks

Virtually all languages use stacks (I can't think of one which doesn't!). The 6502 microprocessor uses one to store return addresses: whenever a JSR is executed the address that the RTS will hand control to is pushed on the stack. A language such as BASIC is designed so that the end-user does not need to know what the stack is doing: it is said, therefore, to be 'transparent' to the user. This is not so in FORTH; moreover, FORTH maintains two stacks. The first FORTH stack is the *Computation* or *Data* stack, which is used to store items of data on which words will operate. The second stack is the *Return* stack, the main use of which is to store the address of the routine to which control is passed after the current word is executed. This is not unlike a standard 6502 stack action.

FORTH uses stacks extensively. Most dictionary entries have some action on the stack. Some expect data to be placed on it before acting on it while others will return results by pushing the result on the stack. In this respect both FORTH stacks are of the standard LIFO (Last In, First Out) kind – in other words, the last item pushed on the stack must be the very first pulled off it.

Implementing these stacks correctly, therefore, is vitally important. The Return stack is straightforward enough, because the standard hardware stack of the 6502 from &1FF down can be used, with the standard PLA and PHA instructions operating on it.

The Computation stack must be a minimum of 64 bytes long and must be stored in zero page: the first 64 locations from &00 to &40 are ideal for this.

Because the Return stack uses the 6502 hardware stack, it has its own readymade stack pointer, the contents of which can be moved to and from the X register with the TSX and TXS instructions.

A stack pointer is also needed for the Computation

or Data stack and it is customary to use the X register for this. As the X register will also be needed by other machine-code routines its contents must be saved and restored from time to time. Its backup temporary store can be a *zero-page* location called XSAVE.

Let us now take a concrete example. In FORTH the word DUP will duplicate the two-byte number currently on the top of the computation stack. The dictionary entry for this primitive would be along the lines of:

```
OPT FNequb (&83)              \ define header
OPT FNequs ("DU")
OPT FNequs (ASC"P"+&80)
OPT FNequw (lastword)         \ link to last nfa
OPT FNequw (dup)              \ CFA address
.dup
LDA 0,X                       \ get first byte. X
                                points to
                                top of
                                computation stack
PHA                           \ save on 'Return'
                                stack (=6502
                                machine stack)
LDA 1,X                       \ get second byte
DEX
DEX                           \ move stack
                                pointer by 2
                                to point to next
                                item
STA 1,X                       \ write to stack
PLA                           \ pull 'saved'
                                result
STA 0,X                       \ write to stack
JMP next                      \ do 'next'
```

The routine assumes that X points to the next free byte on the Computation or Data stack pointer, ie, the next free byte on the computation stack, as indeed it should! The important point to remember with any stack is that it grows downwards, ie, from high location towards low location.

Dictionary format

We have already discussed the format of a dictionary entry: it consists of a header and a body. However, several header formats can be realised and, if space is tight, optimising the header can save considerable space. A common alternative to the 'complete' header already described is the 6-byte approach which is arranged as follows:

Number of characters – 1 byte
ASCII code for first 3 characters – 3 bytes
Link address – 2 bytes

For example, the header for the keyword COMMANDS would be formed as:

```
OPT FNequb (&88)
OPT FNequs ("COM")
OPT FNequw (linkadd)
```

The search routine would need to be adapted to compare a given word with just the first three bytes of the keyword. The length would also need to be tested in such instances – this would allow the interpreter to distinguish between a keyword called COMMANDS and one called COMMA. This is not foolproof, though, because keywords such as DROP and DROT would appear identical. As already mentioned, this 6-byte approach saves dictionary space and this can be quite considerable. For example, a dictionary containing 100 words with a two-byte extra overhead would increase the overall length of the code by 200 bytes.

The two interpreters

A typical FORTH implementation contains two main sections, the *inner interpreter* and the *outer interpreter*.

The inner interpreter is the very heart of the system and is responsible for the ordering of the execution of the machine-code that runs the language. In practice the inner interpreter is composed of three short, fast routines – these are illustrated in figure 8.1.

The outer interpreter forms the interface between the FORTH language and the user. As such it is not written in machine-code (ie, primitives) but in FORTH – it's usually totally composed of secondaries.

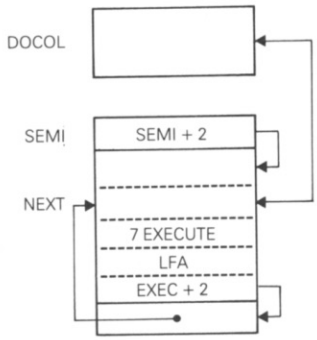


Figure 8.1 Inner interpreter routines. DOCOL jumps to NEXT; SEMI always falls through to NEXT.

To allow FORTH to run a dictionary entry which is composed entirely of addresses, the very first address in the body of the dictionary entry executed *must be that of a routine which will allow this to happen.* In other words, the CFA of the secondary contains the address of a primitive called DOCOL which is short for 'DO COLON definition'. The word COLON has been introduced here and will be discussed later – for now just consider it as a name for an action!

Referring back to figure 8.1 we can see that DOCOL always ends in a jump to NEXT (the NEXT routine performs as already described in the last chapter). The action of DOCOL is roughly as follows.

```
.docol
LDA IP+1        \ get high byte IP
PHA             \ save on Return Stack
LDA IP          \ get low byte IP
PHA             \ save on Return Stack
CLC             \ clear carry flag
LDA W           \ get Word address low byte
ADC #2          \ add 2 to it
STA IP          \ and save it in IP
LDA W+1         \ get high byte
```

```
ADC #Ø          \ add any carry
STA IP+1        \ save it
JMP next        \ and fall through to next
```

As the word must have been called from some other word, IP contains the address of a point in the calling word. The first action of DOCOL is to copy this address from IP on to the Return stack for later retrieval. Two is then added to the contents of W to give the address of the start of the new word's parameter area, and this is stored in IP. Jumping to NEXT will then put the interpretation of the new word into motion.

As in the listing for miniTHORF in the last chapter, all FORTH secondaries end with the address of EXIT. In FORTH, though, this is called SEMI and is a machine code routine that will move the top value on the Return stack into IP and then jump to NEXT. Of course, this value will be the old and original contents of IP put there originally by DOCOL. The code for this is simply:

```
.SEMI
OPT FNequw (cfasemi)
.cfasemi
PLA
STA IP
PLA
STA IP+1
JMP NEXT
```

SEMI is therefore a primitive which will invoke the action of SEMI, and 'fall through' to NEXT, as figure 8.2 shows.

The action of NEXT is identical to that described for the NEXT in miniTHORF. It first extracts the CFA address from the dictionary entry and moves it into W, (the Word Address Register), increments IP by 2, and finally makes an indirect jump to the CFA address. Because of this the CFA of both secondaries and primitives must point to *executable* machine-code.

In the case of a secondary, the machine-code pointed to is that of COLON.

In the case of a primitive the machine-code is in the body of a primitive and follows immediately after the

CFA of the primitive.

Figure 8.2 compares a dictionary entry for a primitive and a secondary in a typical FORTH system.

The only machine-code routines ever executed are primitives and inner interpreter routines such as NEXT. Secondaries may call secondaries that may call secondaries and so on, but at the end of the chain there must always be a primitive with some machine-code to execute!

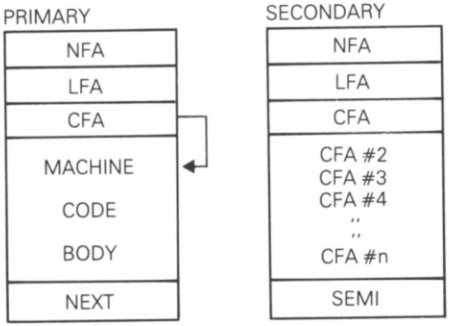


Figure 8.2 Comparison of Primary and Secondary dictionary entries. A primary consists of a head and body – which contains machine code. A secondary consists of a head and a body which contains just addresses which call other dictionary entries.

Referring back to figure 8.1 we can see that the inner interpreter also contains another segment of code. This belongs to the primitive called EXECUTE which is, in fact, the only inner interpreter routine that has a header. This routine is used (as its name implies) to EXECUTE a keyword. The search routine, which belongs in the outer interpreter, searches through the dictionary and, if it locates the specified keyword, it pushes its CFA on the *Computation* or *Data* stack. EXECUTE is then called and pops the address off this stack into W; and jumps *via* W to execute the code pointed to by the CFA address. The dictionary entry for EXECUTE looks like this:

```
OPT FNequb (&87)
OPT FNequw ("EXECUT")
OPT FNequb (ASC"E"+&80)
OPT FNequw (linkaddr)
OPT FNequw (cfaexecute)
.cfaexecute
LDA 0,X                 \ get low byte from
                          stack
STA W                   \ save
LDA 1,X                 \ get high byte from
                          stack
STA W+1                 \ save
INX
INX                     \ adjust stack pointer
                          by 2
JMP W-1                 \ do indirect jump
                            through W
```

Because the Computation or Data stack is being relieved of two items, the stack pointer (held in the X register) must be adjusted by two – which is the purpose of the two INX instructions.

Listing 8.1 illustrates how simple it is to implement a data stack. This program will accept any key you press and push its ASCII value on to a data stack that runs from &43F downwards to &400. The X register is used as a stack pointer and at the start contains &3F. The first entry is pushed on the stack at &400,X or &43F. The X register is decremented by one and so now points to &43E, the next free point on the stack. Whenever a full stop (ASC".") is entered at the keyboard the top item on the stack is pulled and printed on the screen. The X register is incremented by one and now points to the next free point on the stack. As you can see, the stack contents are not wiped out by this process but merely overwritten. Table 8.1 details the main routines in the program.

Routine	Lines	Description
setup	230 to 270	Initialise stack pointer: select Mode 7 screen.
setloop	280 to 320	Print start-up screen
header	330 to 390	Start-up messages
readkey	440 to 550	Read key and print ASCII value. If not '.' then push on stack updating stack pointer
popkey	570 to 620	Pull top of stack and print value, testing for an empty stack
stackfull	640 to 650	Index for Stack Full error message
stackempty	670 to 680	Index for Stack Empty error message
escape	700 to 720	Acknowledge ESCAPE and set index
eloop	740 to 800	Error printing loop
message	820 to 890	Error message buffer

Table 8.1: The main routines of listing 8.1

The important point to remember in any stack operation is that the stack pointer must be tested to see if it is still within range. For example, incrementing X so that it contained &40 would push the next item of data outside the stack. This must be tested for and an appropriate error message reported. Likewise, decrementing X from &00 to &FF should not happen and a Stack Full message must be issued. In both cases, of course, the item of data is 'lost'.

Building dictionary entries

A FORTH program is normally set in motion by entering the name of a keyword which is itself made up of other keywords. For example, we saw earlier that the FORTH word DUP duplicates the top item on the data stack. If we wanted to make two copies of the item on the top of the stack, a word called 2DUP could be defined which would perform DUP twice.

Creating a new dictionary entry requires the formation of a header which is followed by a list of CFAs of the words to be executed.

The simplest *defining word* in FORTH is CREATE, whose action is to *define* a header. For example:

CREATE 2DUP

would create the header for the new word 2DUP. CREATE is usually used with a dictionary entry which has a machine-code body, with the machine-code being poked into the body of the entry after the head.

The commonest way to define a new dictionary entry of a purely secondary nature is to perform a colon definition, typically like this:

```
: 2DUP DUP DUP ;
```

The colon puts FORTH *into compile mode.* Now all words entered will form the body of the dictionary entry for the word being defined, up until a *semi*-colon is encountered. In the case above, the first action when in compile mode is to create the header for 2DUP, and assemble a CFA which points to the code for DOCOL. As you may have guessed, CREATE is used by : (ie, Colon), to do this. The CFA address for DUP will be written after the header, followed by the CFA of DUP again. The semi-colon terminates the formation of the new entry by writing the CFA address of SEMI to finish off the body of the new entry. Now that 2DUP has been added to the FORTH dictionary it may be used in any future word definitions. For instance 3DUP would be created with:

```
: 3DUP 2DUP DUP ;
```

and so on until finally you end up with a single word to do the lot!

Variables and constants

When writing a program in any language, you will invariably use variables and constants. In FORTH the word CONSTANT will create a dictionary entry with a header, but the body of the entry will contain the value specified for the constant. For example, the definition:

```
100 CONSTANT TON
```

will create a header for TON in the normal way and place the value 100 in the body of the entry. In this instance the code field now contains the address of some machine code which uses the address in the word address register, W, to locate the parameter field and copy its contents to the stack, thus:

```
.DOCON
LDY #2
LDA (W),Y          \ get first byte
PHA                \ save on stack
INY
LDA (W),Y          \ get second byte
JMP PUSH           \ goto transfer routine
:
:
.PUSH
DEX
DEX                \ increment stack pointer
STA 1,X            \ save on data stack
PLA                \ pull byte
STA 0,X            \ and save
```

The code ends by falling through to NEXT. The important point to remember here is that W contains the address of the code field of the word currently being executed – and the parameter field (ie, the body) which contains the constant's value is just two bytes further on – which is why the routine begins by setting Y equal to 2.

The structure of a variable is roughly the same as that of CONSTANT. However, for a variable the code field points to some machine-code that uses the contents of W to locate the parameter field and then places its address on the stack. The difference between a variable and constant in FORTH, then, is that when a constant is used its *value* is placed on the stack, whereas when an action on a variable is performed the *address* of the variable placed on the stack. The code that performs this is:

```
.DOVAR
CLC
LDA  W             \ get first byte
ADC #2             \ add 2
PHA                \ save on stack
LDA #0
ADC W+1            \ add any carry
JMP PUSH           \ goto transfer routine
:
```

```
:
.PUSH
DEX
DEX                 \ increment stack pointer
STA 1,X             \ save on data stack
PLA                 \ pull byte
STA 0,X             \ and save
```

As with CONSTANT the code DOVAR falls through into NEXT via a jump to PUSH.

Memory configuration

Figure 8.3 shows a fairly typical memory map for a FORTH type language implemented on a BBC Micro or Electron. We can examine it starting at the bottom and working our way up. At the very base from &00 to &40 (at least) comes the computation or data stack. The stack pointer is the X register and this should have a private storage location along with any other registers, such as IP and W, in the space above this up to &8F. The memory from &90 to &FF is privileged to the MOS and should not be touched. Above this, up to &1FF inclusive, comes the hardware or Return Stack.

The MOS requires memory from &200 to &3FF, but the language is free to use memory from &400 to &7FF for buffers, etc. The value of OSHWM (normally PAGE and typically &1900 on a disc-based BBC Micro) is the ORIGIN of what should be the language proper, containing the inner and outer interpreter plus any other dictionary entries which go to make up your standard TIL system. Above this come any new keywords that you add, known as the Applications, and the top of this is tracked by the Dictionary Pointer, DP. This can grow as far as HIMEM, which will vary depending on the screen mode being used.

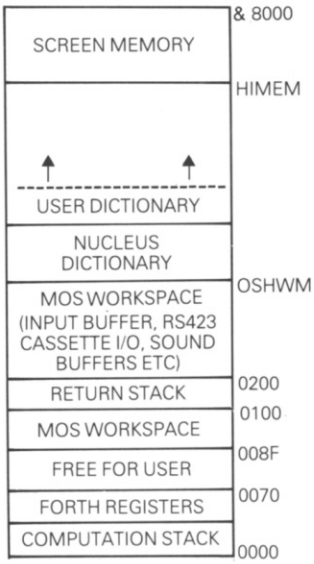


Figure 8.3 A typical FORTH memory map.

Listing 8.1

```
1Ø REM Data Stack Simulation
2Ø REM Mastering Interpreters
3Ø REM and Compilers
4Ø REM (C) Bruce Smith 1986
5Ø :
6Ø PROCvariables
7Ø PROCassemble
8Ø PROCchecksum
9Ø CALL setup
1ØØ END
11Ø :
12Ø DEF PROCvariables
13Ø Xsave=&7Ø:dstack=&4ØØ
14Ø osasci=&FFE3:osrdch=&FFEØ
15Ø osnewl=&FFE7:oswrch=&FFEE
```

```
16Ø osbyte=&FFF4
17Ø ENDPROC
18Ø :
19Ø DEF PROCassemble
2ØØ FOR pass=Ø TO 3 STEP 3
21Ø P%=&5ØØØ
22Ø [OPT pass
23Ø .setup
24Ø LDX #&3F:STX Xsave
25Ø LDA #22:JSR oswrch
26Ø LDA #7:JSR oswrch
27Ø LDX #&FF
28Ø .setloop
29Ø INX:LDA header,X
3ØØ BEQ readkey
31Ø JSR osasci
32Ø BNE setloop
33Ø .header
34Ø OPT FNequw (&ØDØC)
35Ø OPT FNequs ("Acorn MOS")
36Ø OPT FNequw (&ØDØD)
37Ø OPT FNequs ("Data Stack")
38Ø OPT FNequw (&ØDØD)
39Ø OPT FNequb (Ø)
4ØØ \
41Ø .readkey
42Ø LDA #ASC("O"):JSR osasci
43Ø LDA #ASC("K"):JSR osasci
44Ø LDA #32:JSR osasci
45Ø JSR osrdch:JSR osasci
46Ø CMP #27:BEQ escape
47Ø CMP #13:BEQ readkey
48Ø CMP #ASC(".")
49Ø BEQ popstack
5ØØ LDX Xsave:CPX #Ø
51Ø BEQ stackfull
52Ø STA dstack,X
53Ø DEX:STX Xsave
54Ø JSR osnewl
55Ø JMP readkey
56Ø :
57Ø .popstack
58Ø LDX Xsave:INX
```

```
59Ø CPX #&4Ø:BEQ stackempty
6ØØ LDA dstack,X:STX Xsave
61Ø JSR osasci:JSR osnewl
62Ø JMP readkey
63Ø :
64Ø .stackfull
65Ø LDY #&FF:JMP eloop
66Ø :
67Ø .stackempty
68Ø LDY #13:JMP eloop
69Ø :
7ØØ .escape
71Ø LDA #126:JSR osbyte
72Ø LDY #27
73Ø :
74Ø .eloop
75Ø INY:LDA message,Y
76Ø BNE carryon
77Ø JMP readkey
78Ø .carryon
79Ø JSR osasci
8ØØ BPL eloop
81Ø :
82Ø .message
83Ø OPT FNequb(&ØD)
84Ø OPT FNequs("Stack Full!")
85Ø OPT FNequw(&ØØØD)
86Ø OPT FNequs("Stack Empty!")
87Ø OPT FNequw(&ØØØD)
88Ø OPT FNequs("Escape")
89Ø OPT FNequw(&ØØØD)
9ØØ :
91Ø ]
92Ø NEXT pass
93Ø ENDPROC
94Ø :
95Ø DEF FNequb (B%)
96Ø ?P%=B%:P%=P%+1
97Ø =pass
98Ø :
99Ø DEF FNequw (W%)
1ØØØ ?P%=W% MOD 256
1Ø1Ø P%?1=W% DIV 256
```

```
1Ø2Ø P%=P%+2
1Ø3Ø =pass
1Ø4Ø :
1Ø5Ø DEF FNequs (string$)
1Ø6Ø $P%=string$
1Ø7Ø P%=P%+LEN($P%)
1Ø8Ø =pass
1Ø9Ø :
11ØØ DEF PROCchecksum
111Ø X%=Ø
112Ø FOR L%=&5ØØØ TO &5ØCØ
113Ø X%=X%+?L%
114Ø NEXT
115Ø IF X%=22595 THEN ENDPROC
116Ø VDU 7
117Ø PRINT'"Checksum error!"
118Ø STOP
```

CHAPTER NINE

POLISHED POLISH!

TILs such as FORTH, and, as we shall see in later chapters, Compilers, make great use of data stacks when it comes to doing their maths. This, like it or not, is a fundamental part of any language. High-level languages such as BASIC are often quoted as being 'user friendly': what this means is that the language is orientated towards the user and allows him or her to enter programs, expressions and statements in an acceptable form. Acceptable to the user, that is; not the language, which then has the job of sorting the entry into something it can handle. Consider this expression:

9+(3*2)

O-level maths tells us that we must first evaluate any expression held within brackets, so we first perform 3*2=6 and then add 9 to give the result, 15. This is how we would enter such a statement in BASIC, but just consider what the BASIC interpreter has to do. After identifying the line as a mathematical expression it must search it for the presence of brackets, evaluate the contents of the brackets, and then re-evaluate this intermediate result with the rest of the line.

Of course, within the brackets and the rest of the line we also have other levels of hierarchy; for example, a * or / operation will always take precedence over a + or a −. All very user-friendly but a real headache to code!

By rearranging the line we could do away with the brackets: the above example of 9+(3*2) would give the same result if written 3*2+9 but without the brackets. The above example is, of course, very simple, but imagine evaluating a very long expression such as A−(B+C*D)−E/F. Even operations which have the same magnitude of precedence can cause problems. For example, the expression 9−3+2 would give 8. Addition and subtraction are equal in order of import-

ance and so the rule is to evaluate from the left. However, add a pair of brackets to give 9−(3+2) and we obtain a totally different result, 4.

RPN

This fact was noticed in 1921 by a man called Lukasiewicz. His tongue-twisting name lost out to his nationality so his invention is better known as Reverse Polish Notation (RPN). However, his notion was such that the operators always followed the operands (ie, the number or variable itself). In RPN the expression 3*2+9 would be expressed thus:

3 2 * 9 +

The expression is read as usual, from left to right. We scan past the operands to the first operator, the *, and apply this to the two operands immediately to its left, ie, the 3 and the 2, and replace these two with the result and remove the operator. Applying this we get:

6 9 +

Apply the rules once again and we perform an addition on 6 and 9 to give the result 14.

Consider the expression 5+(6*7). In RPN this would be expressed as:

5 6 7 * +

This looks complicated but isn't really. Scan past the operand, moving from left to right; find the first operator, *, and apply it to the two operands to its left, replacing them with the result. Therefore we perform 6*7=42 which leaves,

5 42 +

This gives 5+42=47.

Table 9.1 lists some further examples of RPN. Try substituting real values and working through them.

Standard	RPN
a/b	a b /
a+b*c	a b c * +
a−b−c	a b − c −
(a+b)*c	a b + c *
a/b−c/d	a b / c d / −

a+(b−c/d)−e*f	a b c d / − + e f * −

Table 9.1: Standard versus RPN representation

The RPN stack

In a TIL such as FORTH the use of RPN is fundamental to its operation and all its operations take place on the data stack. When you enter a number in FORTH (or a similar TIL that you may design), it should recognise it as such and push it on the stack. When an operator is entered the TIL should apply it to the top two items on the stack and leave the result on the stack in place of the two operands. Of course the TIL should first assure itself that there are two items to be acted on first, generating any error messages as required. Thus, to add the numbers 7 and 3 together, the following would need to be typed at the keyboard:

```
7
3
+
```

where:

7 is evaluated as a number and pushed on the stack
3 is evaluated as a number and pushed on the stack
\+ is evaluated as an operator and works on the two items on the top of the stack, replacing them with the result.

The result, 10, can now be found on the top of the stack, and could be printed using the print operator, which is . in FORTH (ie, a full stop). We can see what's happening by writing out the contents of the stack after each operation:

action	stack after operation
7	TOP› 7
3	TOP› 3 7
+	TOP› 10

When dealing with any stack-oriented language or compiler it is extremely good practice to write out the stack contents in this way as it helps enormously when it comes to debugging a TIL which is acting up.

Another example:

4 5 6 7 * + + 8 9 * + (or 4+(5+6*7)+8*9)

action	stack after operation
4	TOP› 4
5	TOP› 5 4
6	TOP› 6 5 4
7	TOP› 7 6 5 4
*	TOP› 42 5 4
+	TOP› 47 4
+	TOP› 51
8	TOP› 8 51
9	TOP› 9 8 51
*	TOP› 72 51
+	TOP› 123

Listing 9.1 has listing 8.2 at its heart and uses the data stack principle outlined in the last chapter to perform some RPN. As it stands it will only perform addition, although adding extra operators is very simple. The @ symbol is also recognised and will duplicate the top item on the stack while . will print the top item on the stack.

Enter and run the program which assembles at &5000 and includes a checksum routine. The screen will clear to display the title screen and a prompt. The program will only recognise hexadecimal values less than &FF but the & can be dropped. For example, enter the following items after the OK prompt, pressing

⟨RETURN⟩ after each one:

```
OK  10
OK  10
OK  @
OK  +
OK  +
OK  .
```

Here &10 and &10 are pushed on the stack. The @ will duplicate the top stack item, which will give another &10, so that the stack looks like this:

```
TOP⟩ 10
     10
     10
```

The + will add the top two &10s to leave:

```
TOP⟩ 20
     10
```

The final + will leave:

```
TOP⟩ 30
```

which will be popped off the stack and printed by the .

The new routines used by listing 9.1 are outlined below in Table 9.2.

routine	lines	description
return1	610 to 790	Test for . + and @, branch if required. Try to convert to number, abort if not a number, otherwise push on stack
addthem	810 to 910	Destination of + branch Add top two stack items and push result back on stack
dup	930 to 990	Destination of @ branch Get top stack item and push on stack thereby duplicating it
printtop	1010 to 1070	Destination of . branch Print number on top of stack
stackfull	1090 to 1190	Error message entry points
eloop	1210 to 1270	Print error message
message	1290 to 1350	Error message buffer
convert	1370 to 1490	Call binary to ASCII conversion routine, add results and flag any errors

first	1510 to 1610	Converts accumulator's contents into a binary number. If an illegal character, return with carry set
hexprint	1630 to 1750	Print accumulator's contents as two ASCII bytes

Table 9.2: The new routines in listing 9.1

As it stands, the program will only handle single-byte numbers but could easily be extended to handle the usual two-byte values. Any new operator actions can be added simply by inserting the relevant CMP in the *ready1* routine and inserting the correct coding at a convenient point in the source listing, typically between lines 990 and 1000, renumbering as required.

Listing 9.1

```
 10 REM RPN Stack
 20 REM Mastering Interprters
 30 REM and Compilers
 40 REM (C) Bruce Smith 1986
 50 :
 60 PROCvariables
 70 PROCassemble
 80 PROCchecksum
 90 CALL rpn
100 END
110 :
120 DEF PROCvariables
130 Xsave=&70:dstack=&400
140 param=&80:lbuffer=&8A
150 temp=&8E:linebuffer=&C00
160 oswrch=&FFEE:osasci=&FFE3
170 osword=&FFF1:osbyte=&FFF4
180 osnewl=&FFE7
190 ENDPROC
200 :
210 DEF PROCassemble
220 FOR pass=0 TO 3 STEP 3
230 P%=&5000
240 [OPT pass
250 .rpn
260 LDA #&3F:STA Xsave
270 LDA #22:JSR oswrch
```

```
28Ø LDA #7:JSR oswrch
29Ø LDX #&FF
3ØØ .setloop
31Ø INX:LDA header,X
32Ø BEQ readline
33Ø JSR osasci
34Ø BNE setloop
35Ø .header
36Ø OPT FNequw (&ØDØC)
37Ø OPT FNequs ("Acorn MOS")
38Ø OPT FNequw (&ØDØD)
39Ø OPT FNequs ("RPN Stack")
4ØØ OPT FNequw (&ØDØD)
41Ø OPT FNequb (Ø)
42Ø \
43Ø .readline
44Ø LDA #ASC("O"):JSR osasci
45Ø LDA #ASC("K"):JSR osasci
46Ø LDA #32:JSR osasci
47Ø LDA #linebuffer MOD 256
48Ø STA param
49Ø LDA #linebuffer DIV 256
5ØØ STA param+1
51Ø LDA #128:STA param+2
52Ø LDA #32:STA param+3
53Ø LDA #255:STA param+4
54Ø LDX #param MOD 256
55Ø LDY #param DIV 256
56Ø LDA #Ø
57Ø JSR osword
58Ø BCC return1
59Ø JMP escape
6ØØ \
61Ø .return1
62Ø CPY #Ø:BEQ readline
63Ø \
64Ø LDY #Ø
65Ø LDA linebuffer,Y
66Ø CMP #ASC("."):BEQ printtop
67Ø CMP #ASC("+"):BEQ addthem
68Ø CMP #ASC("@"):BEQ dup
69Ø STA temp:INY
7ØØ LDA linebuffer,Y
```

```
 71Ø JSR convert
 72Ø BCC fine
 73Ø JMP badnumber
 74Ø .fine
 75Ø LDX Xsave:CPX #Ø
 76Ø BEQ stackfull
 77Ø STA dstack,X
 78Ø DEX:STX Xsave
 79Ø JSR osnewl:JMP readline
 8ØØ :
 81Ø .addthem
 82Ø CLC
 83Ø LDX Xsave:INX
 84Ø CPX #&3F:BNE allok
 85Ø JMP stackunder
 86Ø .allok
 87Ø LDA dstack,X
 88Ø INX:ADC dstack,X
 89Ø STA dstack,X
 9ØØ DEX:STX Xsave
 91Ø JMP readline
 92Ø :
 93Ø .dup
 94Ø LDX Xsave
 95Ø CPX #&3F:BEQ stackempty
 96Ø INX:LDA dstack,X
 97Ø DEX:STA dstack,X
 98Ø DEX:STX Xsave
 99Ø JMP readline
1ØØØ :
1Ø1Ø .printtop
1Ø2Ø LDX Xsave:INX
1Ø3Ø CPX #&4Ø:BEQ stackempty
1Ø4Ø LDA dstack,X:STX Xsave
1Ø5Ø JSR hexprint
1Ø6Ø JSR osnewl
1Ø7Ø JMP readline
1Ø8Ø :
1Ø9Ø .stackfull
11ØØ LDY #&FF:JMP eloop
111Ø .stackempty
112Ø LDY #13:JMP eloop
113Ø .escape
```

```
114Ø LDA #126:JSR osbyte
115Ø LDY #27:JMP eloop
116Ø .badnumber
117Ø LDY #35:JMP eloop
118Ø .stackunder
119Ø LDY #47
12ØØ :
121Ø .eloop
122Ø INY:LDA message,Y
123Ø BNE carryon
124Ø JMP readline
125Ø .carryon
126Ø JSR osasci
127Ø BPL eloop
128Ø :
129Ø .message
13ØØ OPT FNequb(&ØD)
131Ø OPT FNerror("Stack Full!")
132Ø OPT FNerror("Stack Empty!")
133Ø OPT FNerror("Escape")
134Ø OPT FNerror("Bad Number")
135Ø OPT FNerror("Stack Underflow")
136Ø :
137Ø .convert
138Ø JSR first:BCC okay
139Ø RTS
14ØØ .okay
141Ø STA &8F:LDA temp
142Ø JSR first
143Ø BCC okayagain
144Ø RTS
145Ø .okayagain
146Ø ASL A:ASL A
147Ø ASL A:ASL A
148Ø ORA &8F:STA &8F
149Ø CLC:RTS
15ØØ :
151Ø .first
152Ø CMP #48:BCC illegal
153Ø CMP #58:BCC ninezero
154Ø SBC #7:BCC illegal
155Ø CMP #64:BCS return
156Ø .ninezero
```

```
157Ø AND #&ØF
158Ø .return
159Ø RTS
16ØØ .illegal
161Ø SEC:RTS
162Ø :
163Ø .hexprint
164Ø PHA
165Ø LSR A:LSR A
166Ø LSR A:LSR A
167Ø JSR dofirst
168Ø PLA
169Ø .dofirst
17ØØ AND #&F:CMP #1Ø
171Ø BCC overnext
172Ø ADC #6
173Ø .overnext
174Ø ADC #48
175Ø JMP oswrch
176Ø ]
177Ø NEXT pass
178Ø ENDPROC
179Ø :
18ØØ DEF FNequb (B%)
181Ø ?P%=B%:P%=P%+1
182Ø =pass
183Ø :
184Ø DEF FNequw (W%)
185Ø ?P%=W% MOD 256
186Ø P%?1=W% DIV 256
187Ø P%=P%+2
188Ø =pass
189Ø :
19ØØ DEF FNequs (string$)
191Ø $P%=string$
192Ø P%=P%+LEN($P%)
193Ø =pass
194Ø :
195Ø DEF FNerror (E$)
196Ø [OPT FNequs (E$)
197Ø OPT FNequw (&ØØØD)
198Ø ]
199Ø =pass
```

```
2ØØØ :
2Ø1Ø DEF PROCchecksum
2Ø2Ø X%=Ø
2Ø3Ø FOR L%=&5ØØØ TO &51BC
2Ø4Ø X%=X%+?L%
2Ø5Ø NEXT
2Ø6Ø IF X%=4562Ø THEN ENDPROC
2Ø7Ø VDU 7
2Ø8Ø PRINT'"Checksum error!"
2Ø9Ø STOP
```

COMPILERS

CHAPTER TEN

INTRODUCTION

Compilers seem shrouded in mystique – special programs which will instantly change your BASIC program into machine-code so that the program will be less demanding in memory requirements, will run much faster and allow you to perform all those real-time tasks you couldn't quite do with BASIC.

Well, that's the normal view and if it's one that you subscribe to, I have some bad news for you! As a general rule, machine-code compiled from a source program will take less room and run faster, but this is not always so. However, the outstanding quality of any compiler is that it will allow you generate machine-code (called stand-alone code) which will run on any BBC or Electron micro regardless of the languages installed. One reason why compiled code will often run faster is not because of the speed of the machine code but because compiled code often has the data which it needs immediately to know, and so does not have to search a sequence of memory locations for it, like interpreted code might. Of course, the compiled machine code is literally just run, so you are executing the action code all the time, unlike an interpreter which must first identify your command before running the appropriate action code.

Compilers, however, do not always compile to machine code. Pascal and BCPL are two languages which are said to be compiled, but they do not compile stand alone-code: they compile to *pseudo-code*. This is a very compact interpreted code, and being compact it takes less space and can run faster. However, such compiled pseudo-code must be interpreted by a pseudo-code interpreter – this can be in the form of a ROM in your machine (as with Acornsoft's Pascal) or it can be incorporated into the compiled pseudo-code. The latter choice will obviously increase (greatly so in

some cases) the memory needed for the entire code, but it does transform it into stand-alone code, although it's pseudo in nature.

Of course, languages such as BASIC store their programs in a tokenised form. This, however, is limited to the keywords which form the language – pseudo code will often tokenise everything including ASCII strings, using just a single byte to represent a few characters.

Such a compiler could be created by extending the principles outlined in previous chapters, using a FORTH-type stack and creating a small but effective tokeniser and interpreter. I'll leave that task to you, as in the next few chapters I'll be providing a compiler which will compile stand-alone machine code from a source program. I'll warn you now, though, that the listing is long – it has to be. However, it is very effective and with some additions to its vocabulary would be comparable to a commercial offering costing much more than this book. Remember, though, that for a small price it is available on the disc that comes with this book.

A parsing phase

The fundamental task of the compiler is to read a source language program written in one language – the source program – and translate it into stand-alone machine-code – the object language. Equally important is that during this period the compiler must check that the source program makes sense; ie, is within the bounds of the compiler's capabilities. If the program exceeds these boundaries the compiler must generate an error and stop or abort compilation.

Classically, compiler actions are divided into a number of phases (described below), each containing a number of sub-phases. In theory each individual phase transforms the source program segment by segment; ie, it takes a piece of the source program and transforms it into the output object code. Because of this step-by-step nature it is possible to define just how each of the separate phases of a compiler are linked together. For example, if each of the separate segments goes through *all* the phases of the compiler before compilation of the next segment then the com-

piler is said to be a *single-pass* compiler – because the object program is compiled in a single pass through the source program. Conversely, if the compiler examined each segment, one after the other, passing through the entire source program several times, then we have a *multi-pass* compiler. However, the two most common types of compiler are the two- and three-pass compilers; that is, the compiler generates the compiled machine code after passing through each segment of the program two or three times. The compiler presented here is a two-pass compiler.

Sticking for the moment with our classical compiler, there are three standard phases, which are:

1) Lexical analysis
2) Syntax analysis
3) Translation

Let's examine each briefly.

Lexical analysis performs a simple translation from ASCII to a token form. This is most commonly done when each line of source program is entered and is not unlike the way in which BBC BASIC operates. Thus, for example, the command PRINT would not be stored internally as five ASCII characters but as a single byte which identifies it as the token for PRINT. The lexical analyser makes the task of the syntax analyser somewhat easier.

The *syntax analyser* determines what the next item of the source program is and directs control to the appropriate point of the compiler. For example, is it a line number? Is it a variable? Is it a label? Is it a command?

Finally, the *translator* does the business of compiling the machine code for the given source program. Figure 10.1 shows the flow diagramatically, and also introduces an extra element, the *symbol table*.

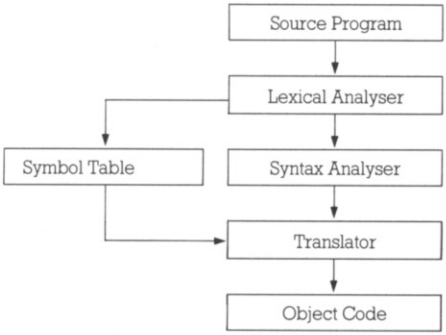


Figure 10.1: The classic compiler

The symbol table is related to the three main stages of the compiler; it contains all the information required by the compiler, for example, the values assigned to variables, etc.

The two L's

The final object code output by the compiler can be structured in two different ways, and the first of these is the *linear construction*. Here the object code is assembled in-line, such that no subroutine calls are used by the object code other than to call standard machine mechanisms (ie, the MOS OS calls). Code generated in this way generally has the advantage of being very fast in execution, but is often very heavy on memory requirements as there is much repetition of machine code; for example, to add two two-byte numbers together.

The second and more common method is to provide a library of machine-code routines which forms the core of the compiler. When the source program is translated into object code the compiler inserts subroutine calls to the library procedures as and when needed. This often greatly reduces the total length of the code. Some compilers have several sets of library procedures and at run-time you can choose which library of routines you wish to use, thus limiting the overall size of the resultant object code.

For example, if the source code contained only integer two-byte arithmetic expressions, it would be rather wasteful to incorporate a library core for use by the compiler that included memory-gobbling floating point routines – these will never be used by the object code. Some clever compilers will construct their own library core using only the bare minimum required for the object code to run correctly. The Library approach may seem rather awkward at first sight, but this is not so ungraceful, not to say tactless, as we shall see!

TACTLESS

Setting out to design a compiler of stand-alone machine-code requires a great deal of thought and general note-making before you even approach the computer. The pre-requisites outlined in the design of Grafrite several chapters ago are all relevant here and there are also many other aspects that must be considered – so let's work through each.

Writing the compiler

There are several ways in which a compiler can be written. We have already decided that our compiler will generate stand-alone machine-code – but there are several ways in which this can be generated:

1) Write it in a high-level language, eg, BASIC
2) Write it in assembly language

The inclusion of procedures and functions in BBC BASIC would make the writing of a linear compiler a relatively simple affair. The source program could be in the form of an ASCII string or a disc-based file or even be incorporated into DATA statements at the end of the compiler that can be read one at a time. Use of IF ... THEN constraints could then be used to call the correct PROC.

For example, a DATA line might READ

```
20000 DATA "MODE",5
```

A condition test would select PROCmode thus:

```
IF command$="MODE" THEN PROCmode
```

This PROC would read the mode number and assemble the machine code thus:

```
10000 DEF PROCmode
10010 READ M%
10020 [OPT pass
10030 LDA #22: JSR oswrch  \ do VDU mode
                           select
10040 LDA #M%: JSR oswrch  \ select mode
10050 ]
10060 ENDPROC
```

The code is assembled in–line, producing the linear compiler. The same approach could be used to assemble a type of library compiler, assembling each PROC-generated code in one area of memory and assembling a JSR to it within the main program elsewhere, using an incrementing value as the label – this is rather messy, though.

A disadvantage of using a high-level language such as BASIC is that the compiler program itself is going to be rather long and leave very little room to assemble the machine code, which by its linearity will also be longish. Of course, the code generation will probably be a little slow, but this is not generally a problem – after all, it is the end result we are after.

The big advantage of writing a compiler in a high-level language with which you are familar is that it is normally a very simple matter to extend it, add to it and modify it. Memory restrictions could also be got round by using a disc overlaying technique whereby sets of PROCs are loaded into memory for use as and when required by the compiler program.

Writing a compiler in assembly code has the advantage in that the resultant machine-code is normally only a fraction of the size of its high-level counterpart, thereby leaving more memory for use by the translator. The resulting code, being more compact, will also be much faster. Use of a library is also infinitely simpler. Use of assembler macros (described below) also keeps the length of the assembly language source code to a manageable length – the library-generating assembly code can also be an independent program, the object code of which is loaded in by the compiler.

The source program to be compiled can also be an independent entity, which makes it much easier to edit if required. Of course, as the compiler itself will be

written in machine-code it can be implemented as a language, which would not be possible using a high-level language approach.

Our compiler will be written in assembly language and contain a machine-code library which will be an independent listing.

The Compiler Source Language

What will the compiler compile? Answering this question will go a long way to defining what commands and operators will be needed in the Compiler Source Language (CSL). The structure here is that of a mini-BASIC but with FORTH-like structures. This means that we need to provide *loop control* and *conditionals*: the BASIC equivalents of these would be REPEAT ... UNTIL loops and IF ... THEN statements. *Subroutines* and *jumps* will also be supported.

Direct commands would include SAVE and LOAD, and a special command to set the translation of CSL to machine-code in motion.

Operators will include standard arithmetic and logical operations on two-byte (ie, 16-bit) numbers.

Variable names may be of any length and may be assigned 16-bit values which may be given in decimal, and *labels* may be used within CSL to mark the destinations of subroutines and jumps. To distinguish CSL commands and CSL variables and labels, all commands must be in upper-case characters, and variables/labels in lower-case characters, ie:

PRINT is a command
print is a variable or label

To distinguish labels from variables all labels must start with a full stop, ie:

number is a *variable*
.number is a *label*

Just by outlining the main aspects of the compiler we have already laid down our ground rules. This information now must be expanded in terms of command names, storage of variable values and label addresses – we can then draw the ever-important memory map.

There is one final aspect to discuss. As we will be using a machine-code library of main subroutines we need a way of transferring data and parameters from the Main Compiled Code (MCC) to the library and back again. This is best done by using a data stack as outlined in the discussion of TILs earlier.

The CSL is, as I have already said, the language to be compiled by the Compiler. The CSL is composed of individual statements and commands, strung together. In BASIC programs these are written in individual lines which can be edited, copied and deleted simply by referencing the number allocated to each line. The Grafrite interpreter held its programs as a long ASCII string – this made editing Grafrite programs difficult without the help of a wordprocessor, and even a minimal change involved quitting Grafrite. The use of line numbers and a small editor is therefore very useful when it comes to editing a program. In our compiler, then, the CSL will be stored as a series of lines each beginning with a line number, and each line can be *listed*, *edited*, *copied* and *deleted* as required. The structure of the line is up for discussion at this stage – the way in which BASIC stores its program lines is ideal, and the same construction will be used, namely:

- two-byte *line number*, high-byte first
- single-byte *pointer* to *start of next line*
- ASCII 13 to *end line*

As before, &FF will be used to mark the very end (ie, TOP) of a CSL program.

Finally, the commands and statements which compose the CSL will be represented internally as single-byte tokens.

Now that the basics of the compiler have been set we can expand on the individual elements and in the process build up the memory map as we go, step-by-step.

The data stack

The data stack will probably be the most used area of memory of the compiler. It's used continually to push items of data on the stack and pull items from it, passing information from the MCC to the library routines and back again. The data stack will only be used for data

and not to hold return addresses – the standard 6502 hardware stack is used for this, in the usual way. Because of the data stack's importance and the flow of data to and from it, the stack will need a reasonable amount of memory, and as such will be allocated 1K, ie, &400 bytes starting at the OSHWM (usually PAGE). Two BASIC variables *stack%* and *stackend%* are used to mark the lower and upper memory boundaries, while the vector *stp* is used by the compiler to hold the stack pointer. Unlike more conventional stacks, the data stack will grow *up* through memory and not down; thus *stp* is loaded with *stack%* on entry. Figure 11.1 shows the first part of the memory map:

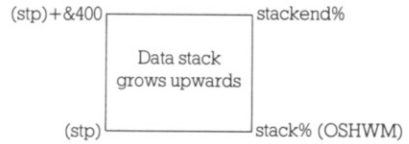


Figure 11.1: Memory map, Part 1 – The Data Stack

Obtaining the value of OSHWM is important, as the data stack needs to be safe from corruption – using memory below this boundary could result in data corruption as the ROM or MOS workspace would be encroached upon. This value can be obtained with an OSBYTE call, ie, with A%=131.

To access the data stack, vectored addresses, or *registers*, will be needed – these will have two purposes and will also act as *pointers*. Of course, at this stage of development you will not be sure how many will be needed in all, so keep names short and meaningful. As these words are being written retrospectively I know that the Compiler needs three such registers: r1, r2 and r3.

We will examine how the data stack is pushed and pulled, when the Compiler Library is discussed in the next chapter.

Library and variable space

The machine-code *library* will be loaded into memory above the stack; and above this the *machine-code* generated by the compiler is loaded – this is obviously variable in length and will be discussed in detail in the next chapter. Immediately above the combined library and compiled code will be the space allocated to store *variable names*, ie, variable names used within the CSL, the first free byte of which will be pointed to by the variable *vspace* which is used by the compiler. The lower end of the memory map is shown in figure 11.2.

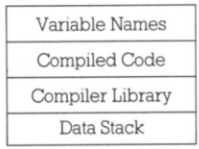


Figure 11.2: Memory Map, Part 2 – The Lower End

The action of the Variable Name space is outlined below.

The Compiler

Because the length of the compiled code and variable name space will vary depending on the length of program, the number of variables used, and so on, the Compiler and CSL programs will be stored at the top end of the memory map, below HIMEM. Of course, the exact location will depend on the screen mode used when the Compiler Assemble source is run. Figure 11.3 shows the arrangement for the top end of the memory map on the assumption that a Mode 6 screen mode has been selected.

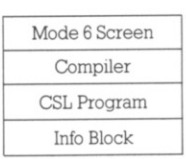


Figure 11.3: Memory Map, Part 3 – The top end. See p.175 for a note on the Info block.

CSL programs will grow in length by shuffling the line numbers down in memory, ie, low line numbers will occupy lower memory locations than high line numbers, as figure 11.4 shows.

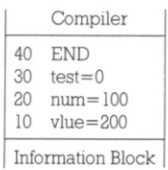


Figure 11.4: Memory Map, Part 4 – CSL program storage

The MCC will compile a main machine-code program, which will simply be a series of JSRs into the library. This is sometimes called *direct threaded code* – as distinct from the indirect threaded code used by FORTH. As an example, the following short CSL program:

```
10 one=100
20 two=200
30 PRINT one+two
```

might compile into the following segement of code:

```
JSR assignvar          \ assign variable value
JSR assignvar          \ assign variable value
JSR getnumber          \ get first number
JSR push               \ push on data stack
JSR getnumber          \ get second number
JSR push               \ push on data stack
JSR add                \ add two numbers on data
                         stack
JSR printnumber        \ print result left on
                         stack
RTS
```

In practice it will not be convenient to handle variables in the way shown above, but the general coding principle is correct.

Variables and labels

The last item in the top end of the memory map is the *information block* which will be used by the Compiler during the translation of the CSL program to the compiled code. Each variable will be allocated five bytes of memory in this block, of which four bytes are used to convey information as follows:

Bytes 1 and 2: Hold an address which points to the variable name within the CSL program.

Bytes 3 and 4: Address of the two bytes allocated to the variable in the space above the compiled code.

Byte 5: For future expansion

Once again, these five bytes are used by the Compiler only *when* it is actually compiling machine code from the CSL program. They are not used by the *final* compiled code. Variables will finally have the initial values assigned to them in the CSL program stored in two bytes within the final compiled machine code. For example, the CSL program might include a line such as:

```
100 test=10
```

In compiled form this would become:

```
JSR getvalue        \ extract two byte number
00
0A                  \ two byte value
JSR xx              \ and continue
```

The two bytes, 00,0A, are the two-byte equivalent of 10 in hexadecimal and are stored directly after a JSR. This JSR calls some machine-code which extracts these two bytes in a similar way to the string printing routine used in earlier chapters, ie, indirectly. If you are confused, remember that in the final compiled code we do not need to know any variable names, just the *values* originally assigned to them! The fifth byte in the variable information is at present unused – it is left for you to use in any way you wish. For instance, you might want to implement single-byte, four-byte or even floating point variables and strings – this fifth byte can be used as a flag to indicate what *type of variable* is being used.

Labels are treated like variables, except no space is allocated to them within the code, and the 3rd and 4th bytes in the 5-byte information block point to an address in the compiled code which relates to the address of the label. Any future references to this label will have this address written into them. For example, a CSL program might have a command called JUMPTO (or GOTO), and as such:

```
10 .start
20 REMARK 1
30 REMARK 2
40 JUMPTO start
```

might compile at address &2000 as follows:

```
2000
2000  JSR remark      \ perform a REM
2003  JSR remark
2006  JMP 2000       \ and JUMPTO start
```

Macros

As mentioned earlier, extensive use of macros is made in the compiler's source code. A *macro* is simply a means of generating frequently used machine-code with a minimum of fuss – we have been using them extensively already, in the form of FNequs, FNequw, FNequd, etc, to simulate BASIC 2's EQUS, EQUB and EQUW which are missing in BASIC 1.

Program 11.1 really shows just how a macro works. It uses a macro to print all 26 letters of the alphabet. The line:

```
OPT FNalphabet (N%)
```

is enclosed inside a loop which also passes the ASCII values of the characters from A to Z into the FN it calls. If you RUN this program you'll see that it generates a large chunk of machine-code which runs from &5000 to &5082. If you look closer you'll see that it loads the ASCII value for each character and calls OSWRCH 26 times! Obviously this program could have been written much more effectively using a smaller machine-code loop, but it does illustrate the principle. The real versatility of

macros comes into play when you have several machine-code routines you need to assemble in-line rather than as a subroutine – then they can be coded as a function and assembled in macro fashion.

Having designed the fundamentals of the compiler we are now ready to tackle the coding of it. Enter TACTLESS, the Truly Amazing Compiled Tokenised Language Enabling Superior Speed!

Listing 11.1

```
10 REM Macro Demo
20 REM Print A to Z
30 REM BBC B/B+/Electron/Master
40 REM (C) Bruce Smith 1986
50 :
60 P%=&5000
70 FOR N%=65 TO 90
80 [
90 OPT FNalphabet (N%)
100 ]
110 NEXT
120 [
130 RTS
140 ]
150 CALL &5000
160 END
170 :
180 DEF FNalphabet (num%)
190 [LDA #num%
200 JSR &FFEE
210 ]
220 =N%
```

CHAPTER TWELVE

THE LIBRARY

The TACTLESS library and Compiler go together – you can't use one without the other.

If you remember that the Library will contain all the standard and frequently used routines you can pretty well write most of it before you even have to think about special routines to include. Again it is simply a matter of sitting down, pen and paper poised, and writing a list of these standard routines and others that you think you will need. Of course, you can simply add new routines to the library as you go – or delete any you aren't using. You will find, though, that you will be developing your Library and Compiler programs together – adding an extra command to the Compiler and then adding extra Library routines to the Library source program. Because of this you will find it helpful to have a good program editor – I am a great fan of the Acornsoft Basic Editor – though it is not a must.

The list

What are the standard routines which will be needed by the library? Here's a list of them:

- input a line of text
- move past spaces
- test byte to see if it's a number
- decimal to hexadecimal conversion
- hexadecimal to decimal conversion
- add two 16-bit numbers together
- subtract one 16-bit number from another
- multiply two 16-bit numbers
- divide one 16-bit number by another
- perform OSCLI
- print an ASCII string of characters
- print number as a decimal string
- test if one 16-bit number is <, > or = to another
- perform logical AND and OR on two 16-bit numbers

That gives you 14 routines for inclusion straight away, drawn from the preliminary Compiler design. We will need to add others, in particular library routines, to manipulate the data stack and also push and pull the three registers, r1, r2, and r3, to and from it. Routines to increment and decrement these registers would be a help. Also, the data stack will need to be reset from time to time. The list of routines has grown to 21 and there will be more as the Compiler progresses.

- set Data Stack
- push two bytes on Data Stack
- pull two bytes from Data Stack
- push r1, r2, r3 on Data Stack
- pull r1, r2, r3 from Data Stack
- increment r1, r2 and r3
- decrement r1, r2 and r3

The LIBINFO file

The TACTLESS compiler needs to have at its disposal a list of subroutine addresses of each of the library routines, so that it can insert the correct JSR *address* when it compiles a TACTLESS CSL program. This means supplying it with a list of library routine names and addresses, which can be done by using a macro. The macro writes the information to a file called LIBINFO when each routine is assembled.

The LIBINFO file can also be used to pass other vital information to the TACTLESS compiler. As the library will sit above the data stack, it must be responsible for calculating the start and end of the stack, so that it 'knows' just where to assemble the library code. This information must also be conveyed to the Compiler via the LIBINFO file, as must the actual length of the library code. This information is written in the first three bytes of the file.

To calculate the length, the library is first assembled over the MOS at &D000 – this does nothing, of course, as you cannot write to a ROM; but it exits with P% containing the end address of the library code as loaded in at &D000. Subtracting this from &D000 gives the length.

Because of the unknown factors involved here, the library program must be capable of relocating itself, moving up memory so that it can assemble the library

code below it.

All this might seem rather convoluted but it's straightforward enough. First assembly, and then the creation of the LIBINFO file, can be done in BASIC. Thus (the line numbers refer to the lines used in listing 12.1):

```
 80 A%=131
 90 hwm=(USR(&FFF4) AND &FFFF00)/256
100 stack%=hwm:stackend%=stack+&400
```

asks the MOS for the address of the OSHWM and calculates the data stack boundaries.

```
140 PROCass (&D000)
150 liblen%=P%-&D000
160 IF PAGE<stackend%+liblen% PROCrelocate
```

This assembles code at &D000 to calculate the length of the library code. If the program is within these bounds, it's relocated above this area of memory.

```
180 out%=OPENOUT"LIBINFO"
190 PRINT#out%,stack%,stackend%
200 PRINT out%,liblen%
210 PROCass(stackend%)
220 CLOSE#out%
```

This creates the LIBINFO file and writes three addresses to it. It assembles code above the data stack and closes the file.

As you can see, the LIBINFO file stays open during the assembly proper of the library, which allows us to write library routine details to it during assembly. This is done by starting each library routine with a macro. As an example, consider the routine to reset the data stack, listed below:

```
.setstp
OPT FNinfo(SETSTACK)
LDA #stack% MOD 256
STA stp
LDA #stack% DIV 256
STA stp+1
RTS
```

The function *FNinfo* is called at the outset and it is this function which writes the details to the LIBINFO file, thus:

```
DEF FNinfo(name$)
IF pass=0 OR library%=&D000 THEN =0
PRINT #out%,name$,P%
=pass%
```

All quite simple; we only want to write to the file if we are on the second assembly pass and we are actually assembling the library code in RAM, thus the test. The name of the routine is written to the file, followed by its address. If SETSTACK was the first library routine in the file and the default of OSHWM was &1C00, then the first few bytes written to the LIBINFO file are shown in figure 12.1 below:

Bytes	Details
00 00 1C 00	stack%=&1C00
00 00 20 00	stackend%=&2000
00 00 03 08	liblen%=&308
53, 43, 54, 53, 54 41 43	ASCII for "SETSTACK"
00 00 20 00	JSR address is &2000

Figure 12.1: Format of LIBINFO file.

Use of the LIBINFO file will be discussed in the next chapter when we examine the working of the TACT-LESS Compiler.

Direct and indirect

As we have already seen, the library will contain routines for pushing and pulling the registers (r1, r2 and r3) to and from the data stack. However, these routines can be performed in two different ways, namely *directly* and *indirectly*, and both will be implemented in the library.

A direct push works by extracting the two bytes after the JSR and pushing these on the data stack. In memory the compiled code looks like this:

```
JSR pushdir      \ push next two bytes
byte1
byte2
JSR xxxx         \ and carry on
```

Here the two bytes represented by *byte1* and *byte2* are extracted and pushed on the data stack, and the library routine *pushdir* to do this is as follows:

```
.pushdir
OPT FNinfo("PUSHDIR")      \ write to LIBINFO
PLA                        \ pull high byte RTS
STA r4                     \ store in temp
                             register
PLA                        \ pull low byte RTS
JSR pushsub                \ move program
                             counter up 2
                             bytes and move
                             bytes into r3
JSR push3                  \ push r3 on Data
                             Stack
JMP (r4)                   \ and jump back
                             after JSR
```

The two subroutines, *pushsub* and *push3*, are other routines within the library.

An indirect push works in a similar way, except that the two bytes following the JSR are treated as an address where the two bytes to be pushed can be found. In this case the compiled code is:

```
JSR pushind          \ next two bytes are an
                       address
byte1                \ low byte address
byte2                \ high byte address
JSR xx               \ next instruction
LDA (byte1),Y
```

Here byte1 and byte2 form an address such that

LDA (byte1),Y

will access the first byte of the two bytes to be pushed on the data stack. The library routine encoding *pushind* is:

```
.pushind
OPT FNinfo ("PUSHIND")      \ details to LIBINFO
PLA                         \ high byte RTS
STA r4                      \ save in temp
PLA                         \ low byte RTS
JSR pushsub                 \ move r4 into r3
LDA (r3),Y                  \ get first byte
PHA                         \ save it
INY
LDA (r3),Y                  \ get second byte
STA r3+1                    \ save at r3+1
PLA                         \ get back byte
STA r3                      \ r3 holds two data
                              byte
JMP pushend                 \ push r3 on Data
                              Stack and RTS to
                              compiled code
```

Both these routines can be seen in the Library source code which is outlined on pages 187–199.

How to enter the library

Program 12.1 is the TACTLESS library. Enter this as listed, but leave out the following lines for the time being:

90, 100, 180, 190, 200, 220, 230 and 5250

and add the following lines:

```
  85 hwm=&4000:stack%=hwm:stackend%=hwm
6000 N%=0
6010 FOR L%=&4000 TO &4308
6020 N%=N%+?L%
6030 NEXT
6040 PRINT N%
```

Save the program before you RUN it! When you do RUN it, messages will be displayed, then the code length will be printed (which should be 776 bytes), and the > prompt will return. The library has been assembled in memory from &4000. Now set the checksum going by typing:

```
GOTO 6000
```

After a few seconds the value 74100 should be printed; if not, sorry, but check your listing!

When you get the correct checksum, delete line 85 and lines 6000 to 6040 inclusive and enter the lines that were first left out. Save your source program using the filename:

```
SAVE "LIBSRCE"
```

and RUN the program. Make sure you have a disc or tape as appropriate, as this filing medium will be brought into action a couple of times. When the prompt reappears, *CAT your disc or tape to see that two new files have been generated, namely:

LIBINFO – the library information file
LIBCODE – The library machine code routines

Make sure that these files are copied on a new disc or tape which will form your TACTLESS Compiler Suite.

The library routines

The main routines in the library are detailed below.

STACKSET: This routine resets the data stack to its default value. The variable *stack%* holds the first location on the stack while the variable *stp* is the stack pointer. Note that the data stack is a two-byte stack, ie, bytes are pushed and pulled in groups of two.

INPUTBUFF: The input line routine. This uses OSWORD 0 and tests for the ESCAPE key. The register r1 is used to point to the input buffer which is pointed to by the variable *input%* and is normally set to &600. The routine ends with a jump to *skipspace*.

SKIP: Uses register r1 as a vector and moves through memory until it encounters a non-space character, ie, any other than ASCII 32 (&20).

CONV: Performs an ASCII string-to-binary-conversion. The ASCII string is pointed to by the address held in r1 and exits with the two-byte binary number pushed on the data stack. Note that this routine will only handle ASCII decimal numbers.

NUM: Tests to see if the byte in the accumulator is within the range ASCII "0" to ASCII "9".

PUSHDIR: Performs a two-byte direct push. The two bytes *following* the JSR instruction in the compiled code are extracted and pushed on the Data Stack.

PUSHIND: Performs a two-byte indirect push. The two bytes after the JSR instruction are used as an *address* at whose location two bytes are extracted and pushed on the data stack.

PULLIND: As PUSHIND, except that two bytes are pulled from the data stack and written to the two-byte address specified after the JSR instruction within the compiled code.

PUSH1: Pushes the contents of r1 on the data stack.

PUSH2: Pushes the contents of r2 on the data stack.

PUSH3: Pushes the contents of r3 on the data stack.

ADD: Performs a 16-bit addition. Expects to find two numbers to be summed on the data stack. The registers r1 and r2 are used to perform the addition, and the result is pushed on the data stack.

SUBTRACT: As for ADD but performs a subtraction. In this case r1 is subtracted from r2, or the bottom number on the stack is subtracted from the top one. (NB 'bottom' means the last one pushed on the data stack).

MULT: As for ADD but a multiplication is performed.

PULL2: The data stack is pulled and deposited in r2.

PULL1: The data stack is pulled and deposited in r1.

PULL3: The data stack is pulled and deposited in r3.

INC1: One is added to the contents of r1.

INC2: One is added to the contents of r2.

INC3: One is added to the contents of r3.

DEC1: One is subtracted from the contents of r1.

DEC2: One is subtracted from the contents of r2.

OSC: Performs an OSCLI command. The address stored after the JSR to this command points to the command line to be issued, which is minus the *.

PRTSTR: Prints an ASCII string terminated by a RETURN character (ASCII 13). The two bytes after the JSR point to the ASCII string to be printed.

DIV: Performs a DIV using the two numbers on the top of the stack. The top number on the data stack is divided by the one below it.

MOD: Finds the MOD of the two numbers on the top of the data stack, as for DIV.

PRNUM: Prints the two-byte number on the top of the stack as a decimal ASCII string.

TRUE: Pushes &FFFF on the data stack to indicate a TRUE condition.

FALSE: Pushes &0000 on the data stack to indicate a FALSE condition.

FLAGTRUE: Logically ORs top two bytes on data stack, thus setting the zero flag in the status register if the number on the stack is false.

<: Tests to see whether first number on data stack is less than the second.

>: Tests to see whether the first number on the data stack is greater than the first.

=: Tests to see if the two numbers on the data stack are equal.

<>:Tests to see if the two numbers on the Data Stack are not equal.

SCREEN: Selects the screen Mode specified by the number on the top of the data stack.

VDU: Pulls the number on the top of the data stack and sends the least significant one to the CDU stream.

AND: Logically AND's the top two numbers on the data stack, leaving the result on the data stack.

OR: Performs a logical ORA between the top two numbers on the data stack, leaving the result on the data stack.

Error messages

The following error messages are generated by the Library:

Escape: The Escape key was pressed during the input of a line.

Stack Overflow: All available space on the data stack has been used. The cure is to allocate more space to the data stack and reassemble Library.

Too Big: The result of an addition or multiplication was too big to fit into just two bytes.

-ve Result: A negative result occurred during a subtraction. Stack Underflow: No more bytes remain on the stack when a pull was performed.

Division by 0: Self explanatory.

Listing 12.1

```
10 REM Tactless Library
20 REM Bruce Smith/Dave Acton
30 REM BBC B/B+/Electron/Master
40 REM (C) Bruce Smith 1986
50 :
60 MODE 7
70 ON ERROR PROCreport
80 A%=131
90 hwm=(USR(&FFF4) AND &FFFF00)/256
100 stack%=hwm:stackend%=stack%+&400
110 PROCinit
120 PROCcentre(2,"Tactless Library")
130 PROCcentre(4,"Calculating code length")
140 PROCass(&D000)
150 liblen%=P%-&D000
160 IF PAGE<stackend%+liblen% PROCrelocate
170 PROCcentre(4,"Code length : "+STR$liblen%+"
    bytes
180 out%=OPENOUT"LIBINFO"
190 PRINT#out%,stack%,stackend%
200 PRINT#out%,liblen%
210 PROCass(stackend%)
220 CLOSE#out%
230 PROCsavecode
240 END
250 :
260 DEF PROCinit
270 DIM os% 4
280 input%=&600
290 osbyte=&FFF4:osword=&FFF1
300 osnewl=&FFE7:oscli=&FFF7
310 osasci=&FFE3:oswrch=&FFEE
320 r1=&20:r2=&22:r3=&24
330 r4=&26
340 stp=&7C:count=&86:temp=&87
350 ENDPROC
360 :
370 DEF PROCass(library%)
380 FOR pass%=0 TO 2 STEP 2
390 P%=library%
400 [OPT pass%
410 .setstp OPT FNinfo("SETSTACK")
420 LDA #stack% MOD 256
```

```
430 STA stp
440 LDA #stack% DIV 256
450 STA stp+1
460 RTS
470 :
480 .inpbuff OPT FNinfo("INPUTBUFF")
490 LDX #pb1% MOD 256
500 LDY #pb1% DIV 256
510 LDA #0
520 JSR osword
530 BCC notesc
540 OPT FNerr("Escape")
550 .notesc
560 LDA #input% MOD 256
570 STA r1
580 LDA #input% DIV 256
590 STA r1+1
600 JMP skipspace
610 :
620 .preskip
630 JSR inc1
640 .skipspace OPT FNinfo("SKIP"
650 LDY #0
660 LDA (i1),Y
670 CMP #32
680 BEQ preskip
690 RTS
700 :
710 .conv OPT FNinfo("CONV")
720 LDA #0
730 STA r2
740 STA r2+1
750 .convloop
760 LDY #0
770 STY r3+1
780 LDA (r1),Y
790 SEC
800 SBC #48
810 STA r3
820 LDA (r1),Y
830 JSR num
840 BCS convout
850 JSR push3
```

```
 86Ø JSR push1
 87Ø JSR push3
 88Ø JSR push2
 89Ø JSR pushdir
 9ØØ OPT FNequw(1Ø)
 91Ø JSR mult
 92Ø JSR add
 93Ø JSR pull2
 94Ø JSR pull1
 95Ø JSR pull3
 96Ø JSR inc1
 97Ø JMP convloop
 98Ø .convout
 99Ø JMP push2
1ØØØ :
1Ø1Ø .num OPT FNinfo("NUM")
1Ø2Ø EOR #48
1Ø3Ø CMP #1Ø
1Ø4Ø EOR #48
1Ø5Ø RTS
1Ø6Ø :
1Ø7Ø .pushdir OPT FNinfo("PUSHDIR")
1Ø8Ø PLA
1Ø9Ø STA r4
11ØØ PLA
111Ø JSR pushsub
112Ø .pushend
113Ø JSR push3
114Ø JMP (r4)
115Ø :
116Ø .pushind OPT FNinfo("PUSHIND")
117Ø PLA
118Ø STA r4
119Ø PLA
12ØØ JSR pushsub
121Ø LDA (r3),Y
122Ø PHA
123Ø INY
124Ø LDA (r3),Y
125Ø STA r3+1
126Ø PLA
127Ø STA r3
128Ø JMP pushend
```

```
129Ø :
13ØØ .pullind OPT FNinfo("PULLIND")
131Ø JSR pull1
132Ø PLA
133Ø STA r4
134Ø PLA
135Ø JSR pushsub
136Ø LDA r1
137Ø STA (r3),Y
138Ø INY
139Ø LDA r1+1
14ØØ STA (r3),Y
141Ø JMP (r4)
142Ø :
143Ø .pushsub
144Ø STA r4+1
145Ø LDY #Ø
146Ø LDX #6
147Ø JSR maininc
148Ø LDA (r4),Y
149Ø STA r3
15ØØ JSR maininc
151Ø LDA (r4),Y
152Ø STA r3+1
153Ø JMP maininc
154Ø :
155Ø .push1 OPT FNinfo("PUSH1")
156Ø LDX #Ø
157Ø BEQ mainpush    .
158Ø .push2 OPT FNinfo("PUSH2")
159Ø LDX #2
16ØØ BNE mainpush
161Ø .push3 OPT FNinfo("PUSH3")
162Ø LDX #4
163Ø .mainpush
164Ø LDA r1+1,X
165Ø LDY #1
166Ø STA (stp),Y
167Ø LDA r1,X
168Ø DEY
169Ø STA (stp),Y
17ØØ LDA stp
171Ø CLC
```

```
172Ø ADC #2
173Ø STA stp
174Ø LDA stp+1
175Ø ADC #Ø
176Ø STA stp+1
177Ø CMP #stackend% DIV 256
178Ø BEQ overflow
179Ø RTS
18ØØ .overflow
181Ø OPT FNerr("Stack overflow")
182Ø :
183Ø .add OPT FNinfo("ADD")
184Ø JSR doublepull
185Ø LDA r1
186Ø CLC
187Ø ADC r2
188Ø STA r2
189Ø LDA r1+1
19ØØ ADC r2+1
191Ø STA r2+1
192Ø BCC push2
193Ø .toobig
194Ø OPT FNerr("Too big")
195Ø :
196Ø .subtract OPT FNinfo("SUBTRACT")
197Ø JSR doublepull
198Ø LDA r2
199Ø SEC
2ØØØ SBC r1
2Ø1Ø STA r1
2Ø2Ø LDA r2+1
2Ø3Ø SBC r1+1
2Ø4Ø STA r1+1
2Ø5Ø BCS push1
2Ø6Ø OPT FNerr("-ve result")
2Ø7Ø :
2Ø8Ø .mult OPT FNinfo("MULT")
2Ø9Ø JSR multdivset
21ØØ .mult2
211Ø ASL r3
212Ø ROL r3+1
213Ø BCS toobig
214Ø ASL r2
```

```
215Ø ROL r2+1
216Ø BCC mult3
217Ø LDA r1
218Ø CLC
219Ø ADC r3
22ØØ STA r3
221Ø LDA r1+1
222Ø ADC r3+1
223Ø STA r3+1
224Ø BCS toobig
225Ø .mult3
226Ø DEX
227Ø BNE mult2
228Ø JMP push3
229Ø :
23ØØ .doublepull
231Ø JSR pull1
232Ø .pull2 OPT FNinfo("PULL2")
233Ø LDX #2
234Ø BNE mainpull
235Ø .pull1 OPT FNinfo("PULL1")
236Ø LDX #Ø
237Ø BEQ mainpull
238Ø .pull3 OPT FNinfo("PULL3")
239Ø LDX #4
24ØØ .mainpull
241Ø LDA stp
242Ø SEC
243Ø SBC #2
244Ø STA stp
245Ø LDA stp+1
246Ø SBC #Ø
247Ø STA stp+1
248Ø CMP #(stack% DIV 256)-1
249Ø BEQ underflow
25ØØ LDY #1
251Ø LDA (stp),Y
252Ø STA r1+1,X
253Ø DEY
254Ø LDA (stp),Y
255Ø STA r1,X
256Ø RTS
257Ø .underflow
```

```
2580 OPT FNerr("Stack underflow")
2590 :
2600 .inc1 OPT FNinfo("INC1")
2610 LDX #0
2620 BEQ maininc
2630 .inc2 OPT FNinfo("INC2")
2640 LDX #2
2650 BNE maininc
2660 .inc3 OPT FNinfo("INC3")
2670 LDX #4
2680 .maininc
2690 INC r1,X
2700 BNE maininc2
2710 INC r1+1,X
2720 .maininc2
2730 RTS
2740 :
2750 .dec1 OPT FNinfo("DEC1")
2760 LDX #0
2770 BEQ maindec
2780 .dec2 OPT FNinfo("DEC2")
2790 LDX #2
2800 .maindec
2810 LDA r1,X
2820 BNE maindec2
2830 DEC r1+1,X
2840 .maindec2
2850 DEC r1,X
2860 RTS
2870 :
2880 .osc OPT FNinfo("OSC")
2890 PLA
2900 STA r1
2910 PLA
2920 STA r1+1
2930 JSR inc1
2940 JSR push1
2950 .oscskip
2960 LDY #0
2970 LDA (r1),Y
2980 JSR inc1
2990 CMP #13
3000 BNE oscskip
```

```
3Ø1Ø JSR pull2
3Ø2Ø LDX r2
3Ø3Ø LDY r2+1
3Ø4Ø JSR oscli
3Ø5Ø JMP (r1)
3Ø6Ø :
3Ø7Ø .printstring OPT FNinfo("PRSTR")
3Ø8Ø PLA
3Ø9Ø STA r1
31ØØ PLA
311Ø STA r1+1
312Ø LDY #Ø
313Ø .printloop
314Ø JSR inc1
315Ø LDA (r1),Y
316Ø CMP #13
317Ø BEQ stringout
318Ø JSR oswrch
319Ø JMP printloop
32ØØ .stringout
321Ø JSR inc1
322Ø JMP (r1)
323Ø :
324Ø .div OPT FNinfo("DIV")
325Ø JSR maindiv
326Ø JMP push2
327Ø .mod OPT FNinfo("MOD")
328Ø JSR maindiv
329Ø JMP push3
33ØØ .maindiv
331Ø JSR multdivset
332Ø LDA r1
333Ø ORA r1+1
334Ø BNE divloop
335Ø OPT FNerr("Division by Ø")
336Ø .divloop
337Ø ASL r2
338Ø ROL r2+1
339Ø ROL r3
34ØØ ROL r3+1
341Ø LDA r3
342Ø SEC
343Ø SBC r1
```

```
344Ø TAY
345Ø LDA r3+1
346Ø SBC r1+1
347Ø BCC nextdiv
348Ø STY r3
349Ø STA r3+1
35ØØ INC r2
351Ø .nextdiv
352Ø DEX
353Ø BNE divloop
354Ø RTS
355Ø :
356Ø .multdivset
357Ø LDA #Ø
358Ø STA r3
359Ø STA r3+1
36ØØ JSR doublepull
361Ø LDX #16
362Ø RTS
363Ø :
364Ø .printnum OPT FNinfo("PRNUM")
365Ø JSR pull1
366Ø .pnum2
367Ø LDX #16
368Ø LDA #Ø
369Ø .pnum3
37ØØ ASL r1
371Ø ROL r1+1
372Ø ROL A
373Ø CMP #1Ø
374Ø BCC less
375Ø SBC #1Ø
376Ø INC r1
377Ø .less
378Ø DEX
379Ø BNE pnum3
38ØØ PHA
381Ø INY
382Ø LDA r1+1
383Ø ORA r1
384Ø BNE pnum2
385Ø .out
386Ø PLA
```

```
387Ø CLC
388Ø ADC #48
389Ø JSR oswrch
39ØØ DEY
391Ø BNE out
392Ø RTS
393Ø :
394Ø .true OPT FNinfo("TRUE")
395Ø JSR pushdir
396Ø OPT FNequw(&FFFF)
397Ø RTS
398Ø :
399Ø .false OPT FNinfo("FALSE")
4ØØØ JSR pushdir
4Ø1Ø OPT FNequw(Ø)
4Ø2Ø RTS
4Ø3Ø :
4Ø4Ø .flagtrue OPT FNinfo("FLAGTRUE")
4Ø5Ø JSR pull1
4Ø6Ø LDA r1
4Ø7Ø ORA r1+1
4Ø8Ø RTS
4Ø9Ø :
41ØØ .lessthan OPT FNinfo("<")
411Ø JSR pull2
412Ø JSR pull1
413Ø LDA r1+1
414Ø CMP r2+1
415Ø BCC true
416Ø BNE false
417Ø LDA r1
418Ø CMP r2
419Ø BCC true
42ØØ BCS false
421Ø :
422Ø .morethan OPT FNinfo(">")
423Ø JSR doublepull
424Ø JSR push1
425Ø JSR push2
426Ø JMP lessthan
427Ø :
428Ø .equal OPT FNinfo("=")
429Ø JSR doublepull
```

```
43ØØ LDA r1
431Ø CMP r2
432Ø BNE false
433Ø LDA r1+1
434Ø CMP r2+1
435Ø BNE false
436Ø JMP true
437Ø :
438Ø .notequal OPT FNinfo("<>")
439Ø JSR doublepull
44ØØ LDA r1
441Ø CMP r2
442Ø BNE true
443Ø LDA r1+1
444Ø CMP r2+1
445Ø BNE true
446Ø JMP false
447Ø :
448Ø .screen OPT FNinfo("SCREEN")
449Ø LDA #22
45ØØ JSR oswrch
451Ø .vdu OPT FNinfo("VDU")
452Ø JSR pull1
453Ø LDA r1
454Ø JMP oswrch
455Ø :
456Ø .and OPT FNinfo("AND")
457Ø JSR doublepull
458Ø LDA r1
459Ø AND r2
46ØØ STA r2
461Ø LDA r1+1
462Ø AND r2+1
463Ø STA r2+1
464Ø JMP push2
465Ø :
466Ø .or OPT FNinfo("OR")
467Ø JSR doublepull
468Ø LDA r1
469Ø ORA r2
47ØØ STA r2
471Ø LDA r1+1
472Ø ORA r2+1
```

```
473Ø STA r2+1
474Ø JMP push2
475Ø :
476Ø .pb1%
477Ø OPT FNequw(input%)
478Ø OPT FNequb(127)
479Ø OPT FNequb(32)
48ØØ OPT FNequb(255)
481Ø OPT FNinfo("@")
482Ø ]NEXT
483Ø ENDPROC
484Ø :
485Ø DEF PROCsavecode
486Ø PROCoscli("SAVE LIBCODE "+STR$~stackend%+"
     +"+STR$~liblen%)
487Ø PROCcentre(19,"Library saved")
488Ø ENDPROC
489Ø :
49ØØ DEF PROCoscli($os%)
491Ø PROCcentre(17,"*"+$os%)
492Ø X%=os%:Y%=os% DIV 256
493Ø CALL oscli
494Ø ENDPROC
495Ø :
496Ø DEF FNequb(byte%)
497Ø ?P%=byte%:P%=P%+1:=pass%
498Ø :
499Ø DEF FNequs(string$):$P%=string$
5ØØØ P%=P%+LENstring$:=pass%
5Ø1Ø :
5Ø2Ø DEF FNequw(word%)
5Ø3Ø ?P%=word%:P%?1=word% DIV 256
5Ø4Ø P%=P%+2:=pass%
5Ø5Ø :
5Ø6Ø DEF FNerr(err$)
5Ø7Ø d%=FNequw(Ø)+FNequs(err$)+FNequb(Ø)
5Ø8Ø =pass%
5Ø9Ø :
51ØØ DEF PROCcentre(y,t$)
511Ø PRINTTAB(Ø,y)SPC4Ø;
512Ø PRINTTAB(2Ø-LENt$/2,y)t$;
513Ø ENDPROC
514Ø :
```

```
515Ø DEF PROCrelocate
516Ø PROCcentre(1Ø,"Relocating program...")
517Ø N%=(stackend%+liblen%+&FF) AND &FFØØ
518Ø VDU 21:*FX 15,1
519Ø *KEYØ FOR X%=(TOP-PAGE) TO Ø STEP
     -1:X%?N%=X%?PAGE:NEXT|MPAGE=N%|MOLD|MRUN|F|M
52ØØ *FX 138,Ø,128
521Ø END
522Ø :
523Ø DEF FNinfo(name$)
524Ø IF pass%=Ø OR library%=&DØØØ THEN =Ø
525Ø PRINT#out%,name$,P%
526Ø =pass%
527Ø :
528Ø DEF PROCreport
529Ø CLS:PROCcentre(2,"Error")
53ØØ CLOSE#Ø:PRINT'':REPORT
531Ø PRINT" at line ";ERL:END
```

CHAPTER THIRTEEN

THE EDITOR

The TACTLESS Compiler is in fact two programs: the Compiler itself and the program Editor. These two programs are very closely linked and share many routines, but during the development stage they should be written separately and then combined to give the finished product. Which one you decide to develop first is a matter of choice; at Sunday lunch I leave the meat till last, so I see no reason to deviate here – the Editor is first! This has an added advantage in that when testing the Compiler it will be possible to edit short test programs written in CSL. As with the Library, you might find that you end up moving freely between the development of both the Editor and Compiler as you adapt and improve. (Listing 16.1, p.262)

As always, let's start by defining what the Editor must be capable of. First and foremost it must be capable of *reading* a line into memory from the keyboard – this is made possible by OSWORD 0 which is a Library function. Once a line is read in, the Editor must place it in memory. However, the Editor must find the correct position for it within the program, as defined by its line number, which will involve moving memory to and fro. The line number must be converted from ASCII decimal to a 16-bit binary value. If a line number only is specified then the relevant line number must be deleted from the program, and again this involves moving blocks of memory. Before we can convert these thoughts into assembler we must first interface the Editor to the Library so that we can take advantage of the latter's subroutines.

Interfacing the library

The LIBINFO file contains the names and execution addresses of all the library routines. The Editor (and Compiler) will need access to these items of data so one of the first things it must do is to read in this

information, which is best read into an array, thus:

```
100 in%=OPENUP("LIBINFO")
110 INPUT #in%,stack%,stackend%
120 INPUT #in%,liblen%
130 lowest=stackend%+liblen%
140 k%=0
150 REPEAT
160 INPUT #in%,lib$(k%),lib%(k%)
170 k%=k%+1
180 UNTIL lib$(k%-1)="@"
```

The first three items of data relate to the stack position and library length. The items after these are routine names and execution addresses which are read into two arrays – *lib$(k%)* contains the library routine names and *lib%(k%)* holds the corresponding execution addresses. The end of the LIBINFO file is marked by an @ and this is used to terminate the array reading-in loop (lines 150 to 180).

To incorporate a subroutine call into the library what's needed is to search the array for the routine name. Once its position in the array is located this will provide the correct index into the address array (ie, *lib%*), enabling the correct JSR to be assembled. All this can be calculated using a suitable function, called from the correct point in the assembler.

As an example, to call the *input a line from the keyboard* routine in the library we would use:

```
OPT FNlib ("INPUTBUFF")
```

where INPUTBUFF is the correct library routine – obviously, it is most important that the name used is the same as the library equivalent used to assemble the LIBINFO file.
FNlib looks like this:

```
11110 DEF FNlib(key$)
11120 addr%=FNlibaddr(key$)
11130 [OPT pass%
11140 JSR addr%
11150 ]:=pass%
```

The JSR to the routine whose address is held in *addr*% is assembled by lines 11130 to 11150. The value assigned to *addr*% is obtained from *FNlibaddr* as follows:

```
11280 DEF FNlibaddr(key$)
11290 k%=-1:REPEAT k%=k%+1
11300 UNTIL key$=lib$(k%) OR lib$(k%)="@"
11320 =lib(k%)
```

A simple loop which compares the sought-after library name with each string in the array is set up. When a comparison is positive the incrementing index is used to pick out the corresponding address in the *lib*% array – thus the correct address is assigned to *addr*% before the JSR being assembled! Use of any library routine is made possible in exactly the same way – just pass its assigned name to *FNlib*. Obviously, a list of the routines is of great value and this can be found in Appendix D, though a detailed description was given in the last chapter.

Line to memory

Having used INPUTBUFF to read a line from the keyboard the Editor must evaluate it. First, the Editor needs to see if the line starts with a number or whether it is a command that needs to be executed straight-away, eg, a * command. On return from INPUTBUFF the accumulator will contain the first non-space character while r1 will point to the address of this, thus:

```
CMP #ASC("*")      \ is it a * command?
BNE nostar         \ branch if not
LDX r1             \ else point X and Y
LDY r1+1           \ to the * command
JSR oscli          \ and pass it to the MOS
JMP tactloop       \ finally re-entering main
                     loop
.nostar
OPT FNlib("NUM") \ test to see if it's a number
                     (0-9)
BCC line           \ carry set if it is, so
                     branch
```

The jump to *tactloop* above simply sets the whole input routine into operation again, by displaying a prompt and then calling INPUTBUFF once more. If a line number is indicated then the branch to *line* takes place and proceeds as follows:

```
OPT FNlib("CONV")   \ convert to 16-bit number,
                      push on stack
OPT FNlib("SKIP")   \ move past any spaces
JSR tokenise        \ tokenise command
OPT FNlib("PULL3")  \ get line number into r3
OPT FNlib("PUSH3")  \ and push back on stack
JSR deleteline      \ test if need to delete a
                      line
LDA input%          \ get next byte
CMP #13             \ is it a blank line?
BEQ tactloop        \ yes; start all again
```

Library routines CONV and SKIP are used to convert the ASCII line number into its two-byte binary equivalent and move on past any spaces. The *tokenise* subroutine when written will evaluate the command and convert it into its equivalent token. At this stage the tokenise routine need do nothing, it can be a straight RTS. The PULL3 routine shifts the two-byte line number currently on top of the data stack into r3, which is then copied back to the stack with PUSH3. The *deleteline* routine tests to see whether the current line number (back on the stack) exists or not. If the line exists this new line will be overwriting or deleting it, otherwise a new line will need to be inserted. The byte at *input%* (next byte in line) is tested to see if it is a return character (and therefore empty) in which case a branch back to *tactloop* can be made; otherwise the new line needs to be added to the program.

```
.insertloop
LDY #0              \ set index
LDA (r2),Y          \ get byte
BMI insertthere     \ branch if TOP
LDA r3+1            \ get low byte line number
CMP (r2),Y          \ are they the same?
BCC insertthere     \ no, greater than; so branch
```

```
BNE lessline     \ no, move to start of program
INY              \ increment index
LDA r3           \ get high byte of line number
CMP (r2),Y       \ is it the same as current
                   one?
BCC insertthere \ no, it's greater than
.lessline
JSR lineskip     \ move to next line number
JMP insertloop   \ and repeat
```

The code bounded by *insertloop* and *lessline* is responsible for testing to see if we are in the correct position to enter the new line. On entry to *insertloop* the register r2 is pointing at the very first program line in memory, while r3 holds the new line number to be inserted. When the correct position in memory is found for the new line, the loop exits by a branch to *insertthere*, otherwise the next line number is located with the help of the subroutine at *lineskip*, and *insertloop* is re-done.

Having found the correct location for the line, within the program, some room must be made for it. This is done by first calculating the length of the new line and opening up a suitable gap in memory for it, taking care not to corrupt any lines already present.

```
LDA page         \ get insert position
STA r1           \ and save in r1
SEC              \ ready for subtraction
SBC len          \ subtract length of line
STA page         \ save new low address
STA r3           \ and save in r3
LDA page+1       \ get high byte
STA r1+1         \ save it
SBC #0           \ subtract any carry
STA r3+1         \ and save high byte
```

At this stage r1 holds the current value of *page* – the base of the program; r3 holds the new value of *page*, ie the value after insertion of the line; and r2 holds the address of the first byte after the space in which the new line is to be put.

```
.movedown
LDA r1 : CMP r2    \ low bytes same?
BNE notmoveend     \ no; carry on
LDA r1+1
CMP r2+1           \ high bytes same?
BEQ moveend        \ yes; all done
.notmoveend
LDY #0             \ get index
LDA (r1),Y         \ get byte
STA (r3),Y         \ and move it
OPT FNlib("INC1") \ increment r1
OPT FNlib("INC3") \ increment r3
JMP movedown       \ and do next byte
OPT FNlib ("INPUTBUFF")
```

Now that the gap has been created the line can be written into it, but first the binary line number and the line-length byte must be deposited into the first three bytes. Note that r3 now points to the gap start.

```
.moveend
OPT FNlib("PULL2")   \ pull line number into r2
LDA r2+1             \ get high byte
STA (r3),Y           \ and write it
INY
LDA r2               \ get low byte
STA (r3),Y           \ and write it
INY
LDA len              \ get line length byte
STA (r3),Y           \ and write it
LDX #255             \ get index
.copyloop
INX:INY              \ bump both indexes
LDA input%,X         \ get byte from input
                       buffer
STA (r3),Y           \ and write it
CMP #13              \ all done?
BNE copyloop         \ no, so repeat
JMP tactloop         \ yes
```

That, then, is the Editor skeleton. However, we did mention a couple of subroutines used by the Editor and one of these, *deleteline*, is quite important, so let's

examine this more closely. The first thing which *deleteline* does is to call another subroutine, *findline*, which, as its name suggests, finds where in memory a specified line number starts, returning the address in r2. It also flags whether it found the line or not, setting the carry flag if the line does not exist, clearing the carry if it does.

```
.deleteline
JSR findline     \ try to locate line
BCS notthere     \ not found, so branch
LDY #2           \ point to line length byte
LDA (r2),Y       \ get line length byte
STA temp         \ save it
.deleteloop
LDA r2+1         \ get high byte address
CMP page+1       \ is it at page?
BNE shift        \ no: so need to move
LDA r2           \ get low byte address
CMP page         \ is it at page?
BNE shift        \ no: so need to move
CLC
ADC temp         \ calculate line length
STA page         \ save result
LDA page+1       \ get high byte
ADC #0           \ and add any carry
.notthere
RTS
```

That ties up the bare essentials of the Editor, which in essence is nothing more than a memory manipulator. However, the transformation of command *string* into command *token* is also an integral part of the Editor and we can now examine the tokeniser mechanism.

The tokeniser

TACTLESS Compiler *source* Language (CSL) programs are stored in RAM in a tokenised form – in other words, each CSL command is not stored as an ASCII string (as with Grafrite) but as a token byte unique to each CSL command. The tokenising process is carried out by the Editor almost immediately after you enter a command line into the input buffer. If you refer back

you will remember that we encountered the

```
JSR tokenise
```

instruction during the Editor discussion.

The tokeniser will need to handle three types of command, which are:

- Commands
- Operators
- Direct Commands

Direct commands are those which need to be executed the moment they are entered at the keyboard; LIST, for example. Operators are commands which perform operations on numeric values, typically logical operators such as OR, AND, MOD and DIV. Finally there are the straightforward commands which will make up the syntax of the language – on which we have yet to decide.

To evaluate the token to be assigned to a command the Editor must have available to it the command string in ASCII form and the token representing the particular command. Another item of information must also be available and that is the address of the command's execution coding, ie, the address to which a JSR is made so as to carry out the machine-code which executes the command's function. To summarise, then, each token table entry is constructed as follows:

⟨Command⟩,⟨token⟩,⟨execution address⟩

If the CSL supports a command called PRINT whose token is &85 and whose execution address is at *print*, the token table entry could be constructed with:

```
FNtok("PRINT",&85,print)
```

where FNtok is a function which simply assembles this information into memory, one piece after the other.

To distinguish between the three types of command, token ranges are used (like those described for BASIC in an earlier chapter) and the ranges are:

Command	&80 to &8F
Operator	&A0 to &AF
Direct	&F0 to &FE

The token &FF is used to mark the top of the token table.

The tokenising process

On entry to the tokenising routine the register r1 points to the source command which needs to be tokenised, while the accumulator contains the first character of the command. The code proceeds as follows:

```
.tokenise
LDY #0                      \ set index to zero
LDA (r1),Y                  \ get first character
                              (again!)
EOR #42                     \ flip bits
STA tokflag                 \ save (negative) result
LDA #(input-1) MOD 256      \ get low byte input
                              buffer-1
STA r2                      \ save in vector
LDA #(input-1) DIV 256      \ get high byte input
                              buffer-1
STA r2+1                    \ save in vector
LDA #3                      \ position of command
                              start in line
STA len                     \ so, set length count
```

The EOR instruction first tests to ensure that the byte in question is not a number, and therefore not a line number, and so tokenising can commence. Having pointed register r2 to the address of the input buffer minus one, the byte *len* has 3 placed into it; *len* is used to keep a count of the current line length. Remember, though, that after it is tokenised the command string will be replaced by a single token byte, therefore affecting its length. We need to keep a count of this change in line length so as to reset the line length byte which comes directly after the line number of each line. Because the first three bytes of a CSL will be the line number (2 bytes) and length byte, we can assume this and start directly from here. The main token searching loop is then entered:

```
.tokenloop
OPT FNlib("INC2")       \ increment r2 by 1
INC len                 \ increment length byte
JSR finkey              \ call subroutine to
                          test command with
                          token table
BCS nottok              \ if carry set then token
                          not found
LDA tokflag             \ get token flag
BEQ nottok              \ if zero then no token
JSR addy                \ add Y to contents of r1
LDA (r3),Y              \ get token from table
LDY #0                  \ clear index register
STA (r2),Y              \ place in input buffer
JMP tokenloop           \ and go again
.nottok
LDY #0                  \ clear index register
LDA (r1),Y              \ get byte from command
 string
STA (r2),Y              \ move up after tokenised
                          line
PHA                     \ save accumulator
OPT FNlib("INC1")       \ increment r1
PLA                     \ restore accumulator
CMP #13                 \ was it end of line?
BNE tokenloop           \ no: so do next byte
RTS                     \ else return
```

The subroutine *findkey* is the main search routine, and what this does is to search through the token table testing the command undergoing tokenisation with the command currently pointed to in the token table. This subroutine is discussed below, but suffice it to say now that if the routine returns with the carry flag set, the token has not been located and the end of the token table has been reached. In such cases a branch to *nottok* is made, with the rest of the CSL line being copied across to *input%*. Otherwise the next entry in the command table is located and the main loop repeated to test this. The *findkey* routine goes like this:

```
.findkey
LDA #vocab% MOD 256    \ get low byte start
                         token table
STA r3                 \ save in vector
LDA #vocab% DIV 256    \ get high byte start
                         token table
STA r3+1               \ save in vector
.findloop
LDY #0                 \ set indexing register
.samekey
LDA (r3),Y             \ get byte from token
                         table
BMI endkey             \ if negative,end of
                         command string
CMP (r1),Y             \ is it same as input
                         buffer?
BNE difkey             \ no: so exit this loop
INY                    \ increment index
BNE samekey            \ and move to next
                         character
.endkey
CMP #255               \ is it end of token
                         table?
RTS                    \ return to calling
                         routine
.notend
INY                    \ increment index
.diffkey
LDA (r3),Y             \ get next byte from
                         token table
BPL notend             \ and branch until
                         negative byte found
TYA                    \ move index count into
                         accumulator
CLC                    \ ready to add
ADC #3                 \ add to 3
ADC r3                 \ and add to r3
STA r3                 \ saving result
LDA r3+1               \ get high byte
ADC #0                 \ add carry
STA r3+1               \ save result
JMP findloop           \ and test next string in
                         token table
```

This subroutine compares the command to be tokenised with a command pointed to in the token table. This test routine is basically no different from the command tables examined in previous chapters. The end of the command string in the token table is marked with the token for that command – the token having its top bit set and therefore being a negative byte. If this is reached the command has been identified and its token located. Otherwise the next command string in the token table must be located and the process repeated. The routine *diffkey* moves r3, the token table pointer, on to the next command string in the token table.

Detokenising

The process of detokenising a line is used by a LIST command, although strictly this comes within the bounds of the TACTLESS Compiler rather than the Editor – but this seems the right place to discuss it. Detokenising is the process by which a token is converted back into its equivalent ASCII command. For example, if the token for a PRINT command is &85 then when a LIST command encounters this, it should identify it as a token, it being greater than an ASCII value. It should then search the token table and print the corresponding command to the screen. The routine responsible for this detokenising is *findtok*. On entry to *findtok*, the accumulator contains the token:

```
.findtok
TAX                     \ save token in X
LDA #vocab% MOD 256     \ get low byte start of
                          token table
STA r3                  \ save it
LDA #vocab% DIV 256     \ get high byte start of
                          token table
STA r3+1                \ r3 points to start of
                          token table

.ftok2
LDY #255                \ set index
.ftok3
INY                     \ increment index
                          register
LDA (r3),Y              \ get byte from token
                          table
```

```
BPL tok3              \ if positive, not token;
                        so branch
CMP #255              \ is it end of token
                        table?
BEQ tokout            \ if yes, branch out -
                        token not found
TAX                   \ move token back to
                        accumulator
CMP (r3),Y            \ is it same as token
                        table?
BEQ tokout            \ yes, branch out
TYA                   \ no, move index into
                        accumulator
CLC                   \ ready to add
ADC #3                \ add to 3
ADC r3                \ add to low byte r3
STA r3                \ and save
LDA r3+1              \ get high byte
ADC #0                \ and add carry
STA r3+1              \ save it
JMP ftok3             \ compare next token
.tokout
RTS
```

On return from this subroutine, if the token has been found, r3 points to the start of the command and this can be printed character by character.

THE NUMBER GAME

Handling numbers

The TACTLESS Compiler must be capable of handling numbers – any two-byte values which are thrown at it. As we are going to make extensive use of stacks it would seem good sense to use the Reverse Polish Notation (RPN) approach outlined in the TIL description a few chapters ago. The RPN handler must also be capable of producing compiled output for the TACTLESS Compiler. For example, when evaluating a numeric assignment it would be helpful if it would also write the value into the compiled code. The RPN handler is decribed below. On entry it will compile the expression stored at the address pointed to by r1.

```
.rpn
OPT FNlib("SKIP")          \ move past any spaces
CMP #13                    \ is it the end of line?
BEQ clearops               \ branch if yes
OPT FNlib("NUM")           \ is it a number?
BCS tryvar                 \ no; try a variable
OPT FNlib("CONV")          \ convert ASCII to
                             16-bit binary
OPT FNlib("PULL2")         \ pull result into r2
OPT FNasslib("PUSHDIR") \ compile code and
                             result!
```

Having located and identified a number, the number is converted from an ASCII decimal string into a 16-bit binary value on the stack, using the Library CONV routine. This value is then pulled into register r2. As we have identified a number (as opposed to a variable) this can be pushed directly into the compiled code after a JSR PUSHDIR instruction – and for this we must use the Compiler *asslib* function which is described below:

```
DEF FNasslib (key$)
addr%=FNlibaddr(key$)
[OPT pass%
LDA#20
JSR compbyte
LDA #addr% MOD 256
JSR compbyte
LDA #addr% DIV 256
JSR compbyte
RTS
]:=pass%
```

It is important to realise that this function is being used as a *macro* and the segment of code it contains is actually assembled 'in-line' with the previous segment of code. The *FNlibaddr* has already been discussed – but, to refresh your memory, it simply searches through the LIBINFO file held in memory to locate the address of the rouhne held in *key$*, which in this case is the address of PUSHDIT, and this is then loaded into *addr%*. The subroutine called *compbyte* takes the contents of the accumulator and writes it to the compiled code. The macro therefore assembles code that will, in effect, write:

```
JSR addr%
```

into the compiling machine code; and *addr%* holds the address of PUSHDIR.

The *compbyte* subroutine is listed below. The vector 'write' points to the next free space in the compiling code:

```
.compbyte
LDY #0              \ clear index register
STA (write),Y       \ place byte in compiling code
INC write           \ move write pointer on
BNE cb2             \ branch if no carry
INC write+1         \ otherwise increment high
                      byte pointer
.cb2
RTS                 \ and return
```

Back to the discussion of the RPN routine. After the code for PUSHDIR has been assembled, the value itself must be written after the *JSR addr%* (refer back to the discussion of PUSHDIR in Chapter 12 if you cannot remember why this is). The two-byte value is contained in register r2 and this is written with the call

```
JSR comp2
```

This makes a dual call to the *compbyte* subroutine to write the two bytes into the compiling code. This gives 5 bytes which convey the following information:

```
JSR pushdir
<low byte value>
<high byte value>
JSR nextcall
```

Continuing with the code for RPN:

```
JSR comp2          \ write two byte value
JMP rpn            \ and do again
```

Of course, the CSL may instead be using a variable to contain the number and the RPN handler must be capable of dealing with this. If a number is not found then a branch to *tryvar* is made:

```
.tryvar
JSR vchar                    \ is it a
                               variable, ie 'a'
                               to 'z'?
BCC tryop                    \ if carry clear
                               then no, so branch
LDA #0                       \ entry parameter is
                               0
JSR var                      \ record address of
                               variable and assign
                               it some space.
OPT FNasslib("PUSHIND")      \ assemble code for
                               JSR pushind
JSR comp2                    \ write address
                               after JSR pushind
JMP rpn                      \ and return
```

As we are dealing with a variable a PUSHIND subroutine call needs to be assembled. After this has been written to the compiling code, the contents of r2 need to be written after it – this is done as before with the *comp2* subroutine. However, these two bytes do not represent the value assigned to the variable, but the *address* of the two bytes of space allocated to the variable. This address was deposited into r2 by the subroutine *var* which will be discussed latter.

A third condition could be found by the RPN handler – it may encounter an *operator* (eg, a multiplication sign, an addition sign, etc). This is handled by the *tryop* routine which itself uses a subroutine called *findop*. As with the token table, an *operator table* must also exist which will allow an operator to be located in a search. In addition to the operator, the operator table also includes information about the operator's precedence and the JSR address which needs to be used to carry out the action specified by the operator. Figure 14.1 lists the operators and their precedence.

Operator	Precedence	Notes
*	8	
CHR$(&A2)	8	DIV token
CHR$(&A3)	8	MOD token
+	6	
–	6	
=	4	
<	4	
>	4	
CHR$(&A4)	4	<> token
CHR$(&A0)	2	AND token
CHR$(&A1)	1	OR token

Figure 14.1: Operator Precedence table

The table shows that we have also included several tokens from commands as these may form part of numeric expressions evaluated by the RPN handler. The higher the number the greater the precedence – thus a * will be performed before an +. The operator table is stored after the token table and is pointed to by *ops%* during assembly.

The *findop* subroutine works as follows:

```
.findop
LDA #ops% MOD 256       \ get low byte address of
                          table
STA r3                  \ r3 will point to it
LDA #ops% DIV 256       \ get high byte
STA r3+1                \ and save
.findoploop
LDY #0                  \ clear index register
LDA (r3),Y              \ get byte from table
CMP #255                \ is it end of operator
                          table?
BNE opcomp              \ no, so branch to
                          continue
RTS                     \ else return
.opcomp
CMP (r1),Y              \ is it the same operator?
CLC                     \ clear carry
BNE nextop              \ no, then test next
                          operator
RTS                     \ else return \ operator
                          found
.nextop
```

The *nextop* routine simply adds 4 to the address in r3 to move it on to the next operator before re-entering at *findoploop*. On return to the calling routine the RPN handler needs to do a variety of actions. One of the most important of these is to maintain an operator stack in case the expression under evaluation is a complex one – or certainly one with more than one operator involved. This operator stack is pointed to by *opstp* and the entry point is the label *assop*:

```
.lowerpri
DEC opstp
JSR assub
.assop          \ enter here
LDY #1          \ point index at precedence
LDX opstp       \ get operator stack pointer
BEQ pushop      \ stack empty so branch
LDA opst%-1,X   \ get previous operator
```

```
CMP (r3),Y          \ compare against new operator
BCS lowerpri        \ it's a lower priority, so
                      swap them
.pushop
LDA (r3),Y          \ greater, so get operator
STA opstp%,X        \ and push on operator stack
INC opstp           \ increment operator stack
                      pointer
INY                 \ increment index
LDA (r3),Y          \ get low byte of JSR address
STA r2              \ save in r2
INY                 \ move to next byte
LDA (r3),Y          \ get high byte of JSR address
STA r2+1            \ and save it
OPT FNlib("PUSH2")  \ push address on data
                      stack
OPT FNlib("INC1")   \ add 1 to r1
JMP rpn             \ start main loop
```

If the operator stack is empty, the operator is pushed directly on it; if an operator is already present on the stack the two are compared and, if required, swapped over so that the one with the greater priority sits on top.

That is how numbers are manipulated by the RPN handler. With a little bit of study and thought it is not too hard to grasp the principles involved. To understand the way in which TACTLESS works and compiles, it will be necessary to understand the function of the RPN handler and also how variables are manipulated.

Variable variables

The main variable handler within TACTLESS is also responsible for setting up labels used within the TACTLESS CSL programs. To recap: TACTLESS variables may be allocated names, and these names can be any combination of ASCII *alpha* characters so long as they are lower-case; therefore:

number=10

is a legitimate variable assignment, but

NUMBER=10

is not.

During compilation each variable is allocated 5 bytes of memory in a special variable table assigned above *page* (see the memory maps in chapter 11). The variable table is pointed to by register r3, while the vector *varpt* points to the next free space in this table.

On entry to the main variable handler the accumulator will contain a Condition flag outlining the action required. The two conditions are:

accumulator=0 read space allocated to variable
accumulator=1 define a variable

The former is only needed during compilation and will be touched on in chapter 16, so for now we will assume that the accumulator contains 1:

```
.var
STA varflag             \ save flag
OPT FNlib("SKIP")       \ move past spaces
LDA page                \ get low byte table pointer
STA r3                  \ put in r3
LDA page+1              \ get high byte table
                          pointer
STA r3+1                \ and save it
.varloop
LDA r3                  \ get low byte table address
CMP varpt               \ is it same as next free
                          byte?
BNE searchforvar        \ no, therefore variable
                          might be there
LDA r3+1                \ get high byte
CMP varpt+1             \ is it the same?
BEQ notfoundbefore      \ yes, so it's the first
                          entry
```

The first action is to place the start of the variable table into r3 – we then need to see if the variable has been defined before. If the values of *varpt* and r3 are the same, the variable has not been encountered before. Indeed, this is the first variable assignment to take place and space must be assigned to it, which is what the *notfoundbefore* routine does. For ease of explanation, I will describe the routine from the *firstpass* entry point, which is several bytes along from this entry point.

The bytes we are missing out deal with the TACTLESS Compiler at run-time and are not really relevant at this time:

```
.firstpass
LDA varflag          \ is it read or write
                       operation?
BEQ skipvar          \ if read, branch
LDA varpt            \ get low byte pointer
SEC                  \ ready to subtract
SBC #5               \ subtract 5
STA varpt            \ save result
STA r3               \ and in r3
LDA varpt+1          \ get high byte
SBC #0               \ subtract any carry
STA varpt+1          \ and save it
STA r3+1             \ don't forget r3
```

What we have done is to extend the variable table by 5 bytes; and we can now write the variable information into these 5 bytes thus:

```
LDY #0          \ clear index
LDA r1          \ get low byte address
STA (r3),Y      \ and write it to first byte
INY             \ move index up
LDA r1+1        \ get high byte address
STA (r3),Y      \ and write second byte
.assign
LDY #2          \ set index to 2
LDA varflag     \ is it a variable or label?
BMI labelonly \ branch if label (ie 255)
LDA vspace      \ get address of allocated space
STA (r3),Y      \ write into table
STA r2          \ and into r2
INY             \ index now = 3
LDA vspace+1    \ get high byte address
STA (r3),Y      \ and write into variable space
STA r2+1        \ and put into r2
```

The variable table has now had the variable's information written into it – this will be needed to complete compilation by the TACTLESS Compiler (discussed in

chapter 16). Figure 14.1 illustrates what has happened so far. Remember that the variable table grows down the memory map; therefore, the subtraction of 5 bytes from the variable table pointer is important. It makes sure that space is created for the information to be written above it:

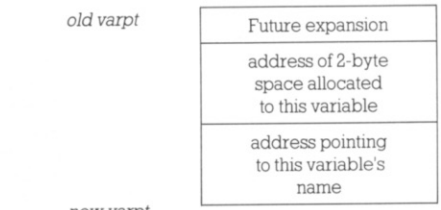


Figure 14.1: Structure of a variable entry in the variable table.

All that now remains to be done is to move the address in *vspace* on by 2. You will recall that *vspace* points to the next free two bytes for the next variable to use as its storage area – in other words, the point in memory where it can hold the two-byte value assigned to it. This simply involves adding 2 to the contents of *vspace*:

```
LDA vspace        \ get low byte
CLC               \ ready to add
ADC #2            \ add 2
STA vspace        \ save result
LDA vspace+1      \ get high byte
ADC #0            \ add any carry
STA vspace+1      \ save result
JMP skipvar       \ now move past variable name
```

The routine here ends with a jump to *skipvar*, which simply increments r1 past the variable name, ending in an RTS back to the calling routine.

In the know

Of course, the variable may have *already* been used in the program and if so will already have some variable

space allocated to it. If this is so, we can use the address already assigned to it – that is, its *vspace* – to store the latest value assigned to the variable. We first need to search through the variable table to see if the current variable already exists or not. This is started by extracting the address of the next variable name and depositing it into a vector called *vcomp*.

```
.searchforvar
LDA r3              \ get low byte
SEC                 \ ready for subtraction
SBC #5              \ minus 5
STA r3              \ save result
LDA r3+1            \ get high byte
SBC #0              \ and subtract any carry
STA r3+1            \ save result
LDY #0              \ clear index
LDA (r3),Y          \ get address from table
STA vcomp           \ and save
INY                 \ bump index
LDA (r3),Y          \ get high byte address
STA vcomp+1         \ and save
DEY                 \ Y=0
```

A simple test-and-compare loop is then entered, the register r1 points to the variable name currently under test and, of course, *vcomp* points to the variable name it is being tested against.

```
.varcomp
LDA (vcomp),Y       \ get byte from variable
                      name
JSR vchar           \ test it's a valid
                      character
BCS foundbefore     \ it's not valid, so two
                      variables are same
CMP  (r1),Y         \ compare against variable
                      name in program
BNE varloop         \ not same, so exit
INY                 \ move index along
BNE varcomp         \ and branch
```

The next byte in *variable name* is loaded via the address in *vcomp*. This address is in the variable table and we are actually accessing a 5-byte variable-block. The *JSR vchar* routine tests to see if the byte is a valid variable character – what this actually does is to tell us when we have reached the end of the variable name – and, if so, the variable has been identified and we can move on to the *foundbefore* routine described below. If it's not the case, the compare is carried out against the corresponding byte in the variable we are trying to identify. If this fails, we go back to *varloop* to try another variable name; otherwise, the index register is incremented by one and the loop repeated. If a variable name is identified through the variable table then control passes to the *foundbefore* routine which then proceeds as follows:

```
.foundbefore
LDA varflag             \ get flag
BNE assign              \ if not zero, branch;
                          otherwise read
LDY #2                  \ index is 2 - point to
                          space alloted
LDA (r3),Y              \ get low byte
STA r2                  \ and save it
INY                     \ Y=3
LDA (r3),Y              \ get high byte
STA r2+1                \ r2 now points to allotted
 space
JMP skipvar             \ move past variable name
```

Here the variable's allotted space pointer is extracted from its entry in the variable table – r2 now contains this and so it is a simple matter to use indirect index addressing – ie, (r2),Y – to read and write the value currently assigned to this variable. We have not yet discussed just how and when this is done – the TACTLESS Compiler does it at run time. It is performed by both the compiler and the CSL programs that manipulate the variables and will be discussed in the next two chapters.

CHAPTER FIFTEEN

THE TACTLESS LANGUAGE

The TACTLESS language is our Compiler Source Language – the program which that will be compiled by the TACTLESS compiler itself. Table 15.1 lists the commands the TACTLESS Language will support and the tokens assigned to each.

Command	Token
IF	&80
CONTINUE	&81
WHILE	&82
NUMERIC	&83
LOOP	&84
PRINT	&85
NOTE	&86
INPUT	&87
END	&88
SCREEN	&89
VDU	&8A
CALLSUB	&8B
JUMPTO	&8C
AND	&A0
OR	&A1
DIV	&A2
MOD	&A3
<>	&A4
LIST	&F9
INFO	&FA
COMPILE	&FB
SAVE	&FC
LOAD	&FD
NEW	&FE

Table 15.1: TACTLESS language commands and tokens

The table shows that 24 commands are available, and of these 11 are true *language* commands, five are *operator* commands and the remaining six are *direct* commands. With the exception of the COMPILE command, each of them is discussed below. The discussion includes both a description of the command's use and the code which carries it out. First the direct commands.

Direct commands

The direct commands are those which are executed directly at the keyboard, ie, they are not normally added to the program. They include NEW, LIST, SAVE and LOAD.

NEW

The function of NEW is to reset the TACTLESS Editor to accept a new TACTLESS language program. Its operation is to *reset* the various system pointers. This is done as follows:

```
.new
LDA #(edstart%-3) MOD 256    \ get low byte
                               Language store
STA page                     \ set page low byte
STA himem                    \ set himem low byte
LDA #(edstart%-3) DIV 256    \ get high byte
                               Language store
STA page+1                   \ set page high byte
STA himem+1                  \ set himem high
                               byte
LDA #255                     \ top marker byte
STA edstart%-3               \ write marker byte
RTS                          \ and return
```

The action of NEW is quite simple: it resets the two TACTLESS pseudo-variables, *page* and *himem*. As the TACTLESS Language program grows down memory (ie, the wrong way) *page* starts with the same value as *himem*, and as new lines are added the value of *page* is decreased – *himem* remains where it is.

LIST

As you would expect, LIST will list any TACTLESS Language program present in memory. A specific

line may also be specified in the LIST command. For example:

LIST

will list an entire program, while:

LIST 10

will list line 10 only (or whatever line number is specified).

The code for LIST is as follows:

```
.list
LDA #(input%+1) MOD 256  \ get low byte of input
                           buffer addr
STA r1                   \ write to r1 low byte
LDA #(input%+1) DIV 256  \ get high byte input
                           buffer address
STA r1+1                 \ write to r1 high byte
OPT FNlib("SKIP")        \ move up past spaces
                           after LIST
OPT FNlib("NUM")         \ test to see if a
                           number is there
BCS listall              \ none: it's a Return,
                           so list all
JMP listline             \ list specified line
.nosuchline
OPT FNerr("No such line") \ error message
.listall
LDA page                 \ get start of low
                           line number
STA r2                   \ save in r2 low byte
LDA page+1               \ get high byte of same
STA r2+1                 \ and save, r2 points
                           to first line
.listloop
LDY #0                   \ set index register
LDA (r2),Y               \ get byte from program
BMI endlist              \ branch if all done
JSR listline             \ and list line number
                           and line
JMP listloop             \ do next line
.endlist
RTS
```

Studying the segment above shows that the *listing* routine is used by both forms of the list process; ie, a total LIST and a LIST with a line number specified. The latter uses a direct JMP to the routine so that the resulting RTS takes the processor back to the calling routine. The former, however, uses this routine as a subroutine, thereby ensuring a return to it so that the next line in the program can be listed. Here the loop is terminated when a negative byte (255) is loaded into the accumulator – this byte is the end-of-program marker. The *listline* routine uses several Library routines:

```
.listline
LDY #0                  \ set index register
LDA (r2),Y              \ get byte (line number
                          high byte)
STA r1+1                \ save in high byte r1
INY                     \ move index on
LDA (r2),Y              \ get low byte line
                          number
STA r1                  \ and save this
OPT FNlib("PUSH1")      \ push r1 on stack
OPT FNlib("PRNUM")      \ print number on stack
OPT FNlib("INC2")       \ increment r2
OPT FNlib("INC2")       \ by 2 past line number
.text
OPT FNlib("INC2")       \ and by 1 again past
                          length byte
LDY #0                  \ clear index
LDA (r2),Y              \ get byte
BPL ord                 \ branch over if not a
                          token
JSR findtok             \ else locate token name;
                          r3 points to it
LDY #0                  \ clear Y again
.disptok
LDA (r3),Y              \ get command name byte
BMI test                \ branch if negative
JSR osasci              \ else print it
CMP #13                 \ was it end of line?
BNE text                \ do next character if not
OPT FNlib("INC2")       \ else move to next line
RTS                      \ and return
```

We discussed the operation of the detokenising routine *findtok* in the last chapter. Note that the line number is first placed into r1 so that it can be pushed on the data stack before being converted to decimal and printed on the screen using the Library PRNUM routine.

INFO The INFO command provides system information and will display both the total number of bytes used by the TACTLESS Language program currently resident and also the number of bytes free. The INFO command coding makes good use of the Library routines, as shown below:

```
.info
OPT FNlib("PRSTR")          \ print following
                              ASCII string
OPT FNequr("Bytes used ") \ ASCII string
OPT FNlib("PUSHIND")        \ push value held at
                              following address
OPT FNequw(himem)           \ assemble address
                              where himem held
OPT FNlib("PUSHIND")        \ push value held at
                              following address
OPT FNequw(page)            \ assemble address
                              where page held
OPT FNlib("SUBTRACT")       \ subtract two numbers
                              on stack
OPT FNlib("PRNUM")          \ print number on top
                              of stack
JSR osnewl                  \ print a new line
OPT FNlib("PRSTR")          \ print ASCII string
                              that follows
OPT FNequr("Bytes free ") \ ASCII string
OPT FNlib(PUSHIND")         \ push value held at
                              next address
OPT FNequw (page)           \ assemble address
                              where page held
OPT FNlib("PUSHIND")        \ push value held at
                              next address
OPT FNequw(lowest%)         \ assemble address
                              where lowest held
OPT FNlib("SUBTRACT")       \ subtract top two
                              numbers on stack
```

```
OPT FNlib("PRNUM")          \ print number on top
                              of stack
JMP osnewl                  \ print new line and
                              return
```

Well, it uses a lot of Library routines and is a very good example of how to use them to construct new commands. The indirect push method is used throughout. To refresh your memory, let's examine what happens in the first PUSHIND operation. This routine extracts the two bytes following this instruction and uses this address as the address of the information. In the first case above the address following that of PUSHIND is that of *himem*. PUSHIND takes the address of *himem* and locates the value stored at the address pointed to by *himem* and pushes this on the stack. The other PUSHIND for *page* and *lowest* work in exactly the same way.

LOAD and SAVE

The LOAD and SAVE commands will save the current TACTLESS Language program. They work as a normal LOAD and SAVE in that a filename must be specified. Their machine-code is not looked at here as it is simply an OSFILE call to save or load, after having extracted the relevant values for address of filename, *page* and *himem*, and deposited them in the OSFILE parameter block.

Commands

We can now examine the main TACTLESS language programming commands, which are discussed in an order that will allow us to examine each in the context of a TACTLESS language program. One point that must be remembered is that when discussing the context of what each command does, we are discussing the operation of what happens at the run time of the Compiler – the program code does not execute the command but defines what happens when the compiled code is *RUN *after* compilation. This should become clearer as you read on. A point to bear in mind is that on entry to all these routines the register `r1` points to the token in the current command line.

NOTE The NOTE command allows comments to be placed within TACTLESS Language programs, and is effectively the equivalent of a BASIC REM command, ie

```
10 NOTE This is a demonstration program
20 NOTE This line is ignored
```

At RUN time the object of the NOTE execution code is simply to move the compiler pointers past a NOTEd line and return control to the Compiler. The current COMMAND under compilation is pointed to by register r1:

```
.note
LDY #255          \ set up indexing register
.noteloop
INY               \ increment index
LDA (r1),Y        \ get byte from line
CMP #13           \ is it end of line?
BNE noteloop      \ no; try next byte
.addy
TYA               \ move index into accumulator
CLC               \ ready to add
ADC r1            \ add end-of-line count to r1
STA r1            \ and save result
LDA r1+1          \ get high byte
ADC #0            \ and add any carry
STA r1+1          \ saving result
RTS
```

When the Compiler identifies a NOTE command a subroutine call is made to *note*; at this stage r1 points to the token for NOTE. Once here, r1 is simply moved up to the end of the program line. The offset from r1 to the end of the line is held in the Y register and this is added to r1 before control is returned to the Compiler. The net effect of this is simply to move the Compiler pointer past the NOTE line and point it at the next line to be compiled. The subroutine *addy* is used extensively by the Compiler.

PRINT The PRINT command can print either a string enclosed in quotes or the value of a variable, and the syntax for each is as follows:

```
10 PRINT "Hello there!"
20 PRINT number
```

Unless a semicolon is placed at the end of the line, a carriage return will be generated, ie,

```
30  PRINT "Value is ";
40  PRINT number
```

The semicolon at the end of Line 30 will ensure that the value of the variable number is placed after the string. In the case of lines 10 and 20 the variables' value will be printed on a new line.

```
.print
OPT FNlib("SKIP")     \ move past spaces after
                        PRINT
CMP #34               \ is it quotes, ie " ?
BEQ string            \ if so, it's a string
CMP #13               \ is it end of line?
BEQ printret          \ yes; so do a return
CMP #59               \ was it a semi-colon?
BNE printv            \ if not, branch
OPT FNlib("INC1")     \ add 1 to r1
LDY #0                \ clear index register
LDA (r1),Y            \ get byte
CMP #13               \ is it a return?
BNE print             \ next character
RTS
```

The *print* routine is therefore simply a test, see, and move-on process, directing control to any particular point required. Let's look at these points in turn; first, the standard string print routine. This is branched to if the result of the first compare is quotes, ie, the start of a string:

```
.string
OPT FNasslib ("PRSTR")  \ assemble: JSR prstr
                          into code
.stringloop
OPT FNlib("INC1")       \ increment r1
LDY #0                  \ clear index register
LDA (r1),Y              \ get byte
CMP #34                 \ is it end quotes?
```

```
BEQ stringout              \ yes: all done
JSR compbyte               \ no: compile byte
CMP #13                    \ is it end of line?
BNE stringloop             \ no; so do next byte
OPT FNerr("Missing """) \ yes; no quotes:
                           signal error!
```

The first action is to compile the instruction:

```
JSR prstr
```

into the compiling code above the Library. The string specified in the quotes is then simply copied to the memory after this JSR. For example, the line:

```
10 PRINT "TACTLESS Compiler"
```

would compile to:

```
JSR prstr
"TACTLESS Compiler"
```

The *compbyte* routine is used throughout to write the current ASCII character in the accumulator into the compiling code. This routine exits when either a closing quote or a simple Return character is found. In the case of the latter, the final quote must have been omitted so an error is generated and the Compiler aborted. Finally, as the line does not contain a semicolon to stop generation of a new line a *JSR osnewl* is compiled. This is set into motion on entry back to the *prin* routine – the accumulator contains 13 so a branch is effected to *printret* which executes the following segment to assemble *JSR osnewl* into the compiling code:

```
.printret
OPT FNlib("PUSHDIR")     \ assemble: JSR pushdir
OPT FNequw("osnewl")     \ assemble: FFE7
.assub
LDA #&20                  \ opcode for JSR
JSR compbyte              \ assemble it
OPT FNlib("PULL2")        \ assemble: JSR pull2
JMP comp2                 \ and compile: FFE7 to
                          give JSR FFE7
```

If a quote, return or semicolon is not encountered by the *print* routine then a branch to *printv* is performed – the assumption being made is that we need to print the contents of a variable or simply a number placed after the PRINT command, ie:

```
50 PRINT number
60 PRINT 123
```

```
.printv
JSR vchar              \ is it a  legitimate
                         character?
BCC okv                \ yes; so branch
JSR rpn                \ evaluate expression
OPT FNasslib("PRNUM") \ assemble: JSR prnum
                         into compiling code
JMP print              \ and return to main loop
```

The RPN handler (as we have already seen) will convert the number from ASCII decimal into a 16-bit value, and write a *JSR pushdir* into the compiling code, following it with the two-byte value. Control will then return and a *JSR prnum* will be compiled to print the number on the top of the stack. A short example should make this clearer. Suppose we had the line:

```
10 PRINT 100;
```

This would compile as follows:

```
JSR pushdir      \ push next two bytes
00 64            \ (100 decimal as two hex bytes)
JSR prnum        \ print number on stack
```

Variables are handled in a similar way, the difference being that a *JSR pushind* is assembled rather than a *pushdir*, and the two bytes which follow are the *address* of the variable's value rather than its actual value. Once again the versatile RPN handler does all the donkey-work for us!

NUMERIC Before a variable can be used within a program it must be defined, ie, an entry must be made for it in the

variable table – a subject covered in the last chapter. To allow this entry to be made, a special variable-defining word exists and it must be used to define all variables thus:

```
10 NUMERIC number
20 NUMERIC counter
30 NUMERIC loop
40 number=0
50 counter=100
60 loop=1
```

The main *var* routine discussed in chapter 14 is used by *numeric* and this is entered with the accumulator containing 1, ie, define variable.

VDU The VDU command allows access to the BBC Micro's main screen driver routines. Basically it takes the value following the command and sends it to the output stream. Numeric values or variables can be used as follows:

```
10 VDU 7
20 VDU 12
```

This when compiled will issue a bell character and clear the screen. It could have been combined into a single line, thus:

```
10 VDU 7,12
```

or alternatively variables could have been used:

```
10 NUMERIC bell
20 NUMERIC clear
30 bell=7
40 clear=12
50 VDU bell,clear
```

Strings of VDU values may also be included as long as each is delimited by a comma, thus 8 forward spaces could be generated thus:

```
10 VDU 9,9,9,9,9,9,9,9
```

The coding is reasonably small:

```
.vdu
JSR rpn                 \ convert and push
OPT FNasslib("VDU")     \ compile: JSR vdu
OPT FNlib("SKIP")       \ move past spaces
CMP #44                 \ is it a comma?
BNE vduout              \ no; so branch out
OPT FNlib("INC1")       \ else move r1 on
JMP vdu                 \ and repeat
.vduout
RTS
```

As with many of the compiling routines the RPN handler does most of the donkey work in writing the VDU parameters into the compiling code before compiling the JSR *vdu*.

SCREEN

This is, as you may have guessed, simply an extension to the VDU command, in that it will generate a mode change. All that needs to be done is to follow SCREEN by the mode number, thus:

```
10 SCREEN 6
```

Of course, this could be performed with the VDU command:

```
10 VDU 22,6
```

The coding for SCREEN simply issues the VDU 22 before sending the specified mode number to the output stream.

LOOP . . . WHILE

This is the TACTLESS Programming language's main loop structure. A LOOP will continue WHILE a particular test remains true. For example:

```
10 NUMERIC count
20 count=0
30 LOOP
40 count=count+1
50 PRINT count
60 WHILE count<100
```

In this case the LOOP . . . WHILE structure is contained within lines 30 to 60, and the loop will continue WHILE the value of *count* remains less than 100. As soon as *count* reaches 100 the loop will exit – what happens then is that the numbers 1 to 100 will be printed to the screen. While this all looks rather complicated it is relatively straightforward to handle – let's examine LOOP first:

```
.loop
LDX loopstp          \ get loop stack pointer
LDA write            \ get current compiling
                       address
STA loopst%,X        \ and write on loop stack
LDA write+1          \ get high byte of address
STA loopst%+1,X      \ and write this to the
                       loop stack
INC loopstp          \ increment loop stack
                       pointer by 2
INC loopstp
RTS                  \ and return
```

Loops are controlled via a *stack* contained in block zero RAM from &540. It is a two-byte stack which is used to hold addresses, and a loop stack pointer (*loopstp*) is kept in zero page at &80. This facility allows loops to be embedded within one another, ie, LOOPs may be bounded by LOOPs so long as each is balanced by a WHILE, thus:

```
50 LOOP
60 LOOP
70 PRINT inner
80 WHILE inner<100
90 PRINT outer
100 WHILE outer<5
```

Up to 20 LOOP . . . WHILE structures may be active at any time. The action of the loop coding, therefore, is simply to push the current *write* address on the stack – *write* points to the next free byte in the compiling code.

```
.while
JSR rpn                    \ evaluate expression
OPT FNasslib("FLAGTRUE") \ compile code to test
                           true flag
LDA #&F0                   \ opcode for BEQ
JSR compbyte               \ compile this
LDA #3                     \ and compile 3
JSR compbyte               \ to give BEQ P%+3
LDA #&4C                   \ opcode for JMP
JSR compbyte               \ compile it
DEC loopstp                \ decrement loop stack
DEC loopstp                \ pointer by 2
LDX loopstp                \ get pointer into X
LDA loopst%,X              \ get byte from stack
JSR compbyte               \ and compile it
LDA loopst%+1,X            \ get high byte
JMP combyte                \ and compile it
```

The *while* code compiles two important instructions into the compiling code. These are:

```
BEQ P%+3
JMP loopaddr
.P%+3
```

If the expression under evaluation succeeds then a true flag will be left on the data stack. This is tested by the

```
JSR flagtrue
```

compiled initially by the *while* code. If the flag is true then the BEQ will be executed and the JMP instruction bypassed. However, if a false flag is left on the stack the BEQ will not take place, and the JMP will be performed back to the address pushed on the loop stack by the corresponding LOOP instruction. As you can see, it is important that all LOOPs are balanced by corresponding WHILEs to keep the compiled code operating correctly. As a programmer this is *your* responsibility!

END The END command simply compiles an RTS instruction into the compiling code; its purpose is to return control to the calling routine. As such, it is important to include it in your programs to make sure that the compiled

code RTS's correctly. You can include as many ENDs as required, and the command coding is simply:

```
.rts
LDA #&60          \ RTS opcode
JMP compbyte      \ and compile
```

CALLSUB This command allows subroutines to be called. Subroutines may be given names, but note that these are in the form of *labels*. A label is simply a variable preceded by a full stop. Like variables, labels may only contain lower-case characters. CALLSUB commands are always terminated with an END command – this effectively performs an RTS at machine level to the calling routine. The following short program will print the numbers 1 to 100 using a subroutine:

```
10 NOTE CALLSUB Demo
20 NUMERIC counter
30 counter=0
40 LOOP
50 counter=counter+1
60 CALLSUB output
70 WHILE counter<100
80 END
90 .output
100 PRINT counter
110 END
```

The END in line 110 is important, as it returns control back to the main program after the CALLSUB command in line 60. So is the END in line 80 which prevents the routine falling through to line 90.

The CALLSUB coding makes full use of the *var* routine (discussed in the last chapter), entering with the accumulator containing 0 to read the space allocated to the label – this space contains the address of the label within memory to which a JSR is compiled:

```
.callsub
LDA #0              \ read space allocated to
                      label
JSR var             \ do it! Address returned in
                      r2
```

```
OPT FNlib("PUSH2") \ push r2 on stack
JMP asssub         \ and compile: JSR (r2)
```

JUMPTO The JUMPTO command also uses a label to define where the program should JUMPTO, ie:

```
10 NOTE Simple JUMPTO Demo
20 JUMPTO skip
30 NOTE miss this
40 NOTE and this
50 .skip
60 END
```

The coding for JUMPTO is very similar to CALLSUB except that a JMP instruction is compiled instead of a JSR, as follows:

```
.jumpto
LDA #0
JSR var       \ read lable addres into r2
LDA #&4C      \ JMP opcode
JSR compbyte \ compile it
JMP comp2     \ and compile address in r2
```

INPUT The INPUT command allows numbers to be entered into your programs – ASCII strings cannot be entered, though you could, of course, adapt the program to accept them. Values being input are read into a specified variable which must first be declared. The following TACTLESS Language program illustrates this:

```
10 NOTE INPUT Demo
20 NUMERIC value
30 PRINT "Enter a number:";
40 INPUT value
50 PRINT "You entered: ";
60 PRINT value
```

The machine-code for the INPUT command is quite short but the longest in terms of the resulting code compiled. Here it is:

```
.input
LDA #0                          \ want to read
                                  variables address
JSR var                         \ read address and
                                  push on stack
OPT FNasslib("INPUTBUFF") \ compile: JSR to read
                                  keyboard
OPT FNasslib("CONV")        \ compile: JSR to
                                  convert number
OPT FNasslib("PULLIND")     \ compile: JSR to pull
                                  indirect
JMP comp2                       \ compile 2 bytes in
                                  r1
```

To get a more detailed view of what's happening, it would be worth referring back to chapter 12 where the Library procedures are discussed; but, to summarise:

First, the *JSR var* with the accumulator containing 0 results in the variable specified in the INPUT statement having its alloted-space address being read and deposited into register r2.

Then, the code for reading the input buffer is assembled into the compiling code – this will read in an ASCII string, and the code compiled after it will result in a subroutine call to the Library-based ASCII decimal-to-binary conversion routine, with the result being deposited on the stack. The JSR for *pullind* is compiled – this takes the two bytes following and stores the data on the stack at this address.

Finally, the two bytes which form the address – ie, the variable's address deposited on the stack originally by *JSR var* – is compiled.

What is actually compiled by *input*, then, is:

```
JSR inputbuff     \ read keyboard
JSR convert       \ convert to binary and push
                    on stack
JSR pullind       \ place number on stack at
                    following address
xx zz             \ two byte address of
                    variable
```

IF . . . CONTINUE

This is the TACTLESS Language condition test structure. Those familiar with FORTH may well recognise that it is not too dissimilar in operation to FORTH's IF . . ELSE . . THEN construct. In simple terms, IF a condition or expression is found to be true then all the commands up to the occurrence of the CONTINUE will be executed. On the other hand, IF the condition or expression is false then the program executes from the CONTINUE command, missing out all the commands in between. The following demonstration program illustrates the principle:

```
 10 NOTE If...Continue Demo
 20 NUMERIC count
 30 count=1
 40 .loop
 50 IF  count<100
 60 PRINT "Count=";
 70 PRINT count
 80 count=count+1
 90 JUMPTO loop
100 CONTINUE
110 PRINT "All done!"
120 END
```

The IF test is situated in line 50, and will be true for the time that *count* remains less than 100, so lines 60, 70, 80 and 90 will be executed. On reaching 100, the IF test will be false and so control will be passed to the CONTINUE command in line 100. The effect of this is that the value of *count* is printed from 1 to 99 and the program ends with the string "All done!" being printed.

As with the LOOP . . WHILE structure, a stack is utilised so that IF . . CONTINUEs may be embedded within one another – again, just make sure that each IF has a corresponding CONTINUE. The coding for IF is as follows:

```
.if
JSR rpn                    \ evaluate expression
OPT FNasslib("FLAGTRUE") \ compile: JSR flagtrue
LDA #&D0                   \ opcode for BNE
JSR compbyte               \ compile opcode
```

```
LDA #3                  \ branch over JMP
JSR compbyte            \ BNE P%+3 now
                          compiled
LDA #&4C                \ opcode for JMP
JSR compbyte            \ compile it
LDX ifstp               \ get if stack pointer
LDA  write              \ get current compile
                          address low byte
STA ifst%,X             \ save on stack
LDA write+1             \ get current compile
                          address high byte
STA ifst%+1,X           \ save it on stack
INC ifstp               \ increment stack
                          pointer by 2
INC ifstp
JSR compbyte            \ compile a dummy byte
JMP compbyte            \ compile another
                          dummy byte
```

The first segment of the above code should be familiar from the WHILE coding – an expression needs to be evaluated and a BNE is compiled to branch over the following JMP when a true condition is signalled. However, unlike the WHILE command, the IF command has to evaluate its expression *on entry rather than on exit* – and at the moment it does not know where the CONTINUE address (ie, the JMP) will be to. For this reason two dummy bytes are compiled into the compiling code – these will be written to when the CONTINUE command is encountered. The address of the position of these two dummy bytes is pushed on the IF stack. When CONTINUE is encountered it proceeds as follows:

```
DEC ifstp           \ decrement IF stack pointer
                      by 2
DEC ifstp
LDX ifstp           \ get stack pointer
LDA ifst%,X         \ get low byte write address
STA r3              \ save it
LDA ifst%+1,X       \ get high byte write address
STA r3+1            \ r3 now contains address of
                      dummy bytes
```

```
LDY #0          \ clear indexing register
LDA write       \ get low byte of current
                  CONTINUE addr
STA (r3),Y      \ write it into low byte dummy
INY             \ Y=1
LDA write+1     \ get high byte CONTINUE
                  address
STA (r3),Y      \ write into high byte dummy
                  address
RTS             \ and return
```

What CONTINUE does is to pull the address of the dummy bytes off the IF stack. Into this it then writes the current contents of *write*, ie, the next free byte into which code can be compiled. When this is done the JMP address will contain that of the next compiled code after the CONTINUE. CONTINUE does not compile any other code; it doesn't need to, as it just informs IF where it needs to JMP to – if and when the expression under evaluation becomes true!

Operators

The only other item to discuss here is the way in which the operators are compiled. Because most of the operators are fundamental to the operation of any reasonably long piece of machine code, (ie, add, subtract, multiply and divide) it should not be too much of a surprise to learn that these routines all compile JSRs into the Library. To do this a function is employed, FNop, which compiles the correct JSR into the compiling code.

CHAPTER SIXTEEN

THE COMPILER

The TACTLESS Compiler mechanism is set into action by the direct command COMPILE. This mechanism is responsible for translation, ordering, interfacing the Library and saving the stand-alone machine code created from a TACTLESS Language program. The Compiler also has to trap errors if and when it encounters them.

As mentioned before, the TACTLESS Compiler is a two-pass compiler and each pass has a vital role. In the first pass the Compiler is becoming 'spatially' aware of itself, calculating addresses and storing them. For example, we saw in the last chapter how the JUMPTO or CALLSUB commands work. On a first pass the Compiler is unable to assign a JMP or JSR address to a *forward* label, ie, one it has not yet encountered. As each label is encountered its name and address are saved into the appropriate bytes within the variable space. In the second pass the addresses are known, and so can be read from the variable space and *assigned as the operands* to the JSR or JMP instructions. The two passes can best be called Analyse and Translate, as illustrated in figure 16.1.

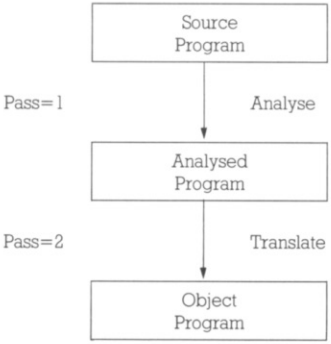


Figure 16.1: An idealised view of a two-pass compiler.

We have already seen how all the subroutines built up around the TACTLESS Compiler work, so let's now examine the coding of the TACTLESS Compiler and learn its darkest secrets . . .

The first pass

Not surprisingly, the first task undertaken on entry to the main *comp* routine is for the various memory pointers and counters to be set-up. These include the pass counter, *pass*; *varpt*, a pointer to a list of variable names; and *vspace*, the address of the next byte to be allocated to variables in the variable space. The various operation stacks are also cleared ready to accept data:

```
.comp
LDA #2                  \ it's 2 passes!
STA pass                \ keep track of current
                          pass
LDA page                \ get start of free
                          space
STA varpt               \ and use this for
                          storing
LDA page+1              \ the addresses of
                          variable
STA varpt+1             \ names
LDA #lowest MOD 256     \ place the next free
                          location
STA vspace              \ at the 'bottom' of
                          memory
LDA #lowest DIV 256     \ in vspace for use as
                          variable
STA vspace+1            \ storage space
.passloop               \ main pass loop
LDA #0                  \ clear accumulator
STA opstp               \ clear operator stack
                          pointer
STA loopstp             \ clear LOOP stack
                          pointer
STA ifstp               \ clear IF stack pointer
LDA page                \ get address of first
STA r1                  \ line in the TACTLESS
```

```
LDA page+1                  \ Language program that
                              is
STA r1+1                    \ to be compiled
LDA #lowest MOD 256         \ get address of first
                              free
STA write                   \ byte into which the
LDA #lowest DIV 256         \ first machine code can
STA write+1                 \ be compiled
OPT FNasslib("SETSTACK")    \ compile code to: JSR
                              setstack
```

The register r1 is used to point to the command currently being interpreted by the Compiler, and as we have seen, the vector *write* contains the address into which the next section of compiling code can be written.

The first operation in any compiled code must be for the data stack to be initialised, and thus the instruction

```
JSR setstack
```

is always the *first instruction* compiled above the Library.

We then enter into the main inner loop of the Compiler. The first action here is to get the first token from the program and identify it. Of course, other checks must also be made here because on future passes we will encounter variables, labels and even the end of the program, and these must all be tested for:

```
.passloop2
LDA r1                      \ get current position in
                              program
STA cur                     \ and save it in 'cur'
                              low byte
LDA r1+1                    \ get the high byte
STA cur+1                   \ and do the same
LDY #0                      \ set index to zero
LDA (r1),Y                  \ get high byte line
                              number
BMI nextpass                \ branch if it is end of
                              the program
LDY #3                      \ point past line details
LDA (r1),Y                  \ get first item in line
```

```
                        - so branch
JSR vchar               \ is it a variable?
BCC varcomp             \ yes; so branch and
                          process it
CMP #46                 \ is it a ".", ie a label?
BNE compstar            \ no; so move on
INY                     \ yes: move to first
                          character
JSR addy                \ add Y to contents of r1
LDA #255                \ define label code
JSR var                 \ process label
JMP checkend            \ line done!
```

So, the first segment of code here points to the beginning of the line. This will normally contain the high-byte of the line number. If the end of the program has been reached then this will be the end-of-program marker 255 (&FF), and the BMI operation tests for this. Loading Y with 3 then moves r1 past the first 3 bytes, which are the same for all lines – and now there is access to the very first programmed character. In most instances this will be a command represented by a token. As all tokens are numbers above &80 the BMI will result in a branch to *comptok*. The only other two conditions possible are that a variable assignment is taking place (tested by the JSR to *vcahr*) or a label is being defined (tested for by identifying a full stop). The routines branched to by the segment of code above are outlined in the next couple of pages:

```
.compvar                \ test variable assignment
JSR addy                \ adds Y to contents of r1
JSR set                 \ process line
JMP checkend            \ and carry on
```

The *compvar* routine is responsible for controlling the setting of variable values, in particular the assigning of one variable value to another, for example:

```
number=number+2
```

would be processed when *compvar* uses the subroutine labelled *set*, thus:

```
.set
LDA #0
JSR var                 \ read address of variable
OPT FNlib("PUSH2")      \ push it on stack
OPT FNlib("SKIP")       \ move past spaces
CMP #61                 \ is it an equals?
BEQ equals              \ branch if yes
JMP mistake             \ no; signal error
.equals
OPT FNlib("INC1")       \ move past =
JSR rpn                 \ evaluate expression
OPT FNlib("PULL2")      \ pull back address
OPT  FNasslib("PULLIND")  \ compile code to:
                            JSR ind
JMP comp2               \ compile contents of r2
```

If an equals sign is found the routine first evaluates and then compiles a *JSR indir* followed by the address of the two-byte result of the expression. If on the other hand the operator is not an equals sign, an error is generated.

The *compstar* routine is used to compile a star command used within a program, for example:

```
20 *GDUMP
```

where *GDUMP might be a graphics screen dump routine to a printer. Effectively a *JSR osc* is compiled into the code, followed by the ASCII string which is to be passed across into the command line interpreter:

```
.compstar               \ test for * command
CMP #ASC("*")           \ is it a star?
BNE mistake             \ no: signal error
JSR addy                \ add Y to contents of r1
OPT FNasslib("OSC")     \ compile code to perform
                          Oscli
.compstarloop
OPT FNlib("INC1")       \ increment r1
LDY #0                  \ index is zero
LDA (r1),Y              \ get byte of command
JSR compbyte            \ compile byte into
                          compiling code
```

```
CMP #13                 \ end of command line?
BNE compstarloop        \ no; do next byte
BEQ checkend            \ yes; so carry on
```

Comptok is responsible for finding the execution coding which carries out the action specified by the current token. The routine calls another subroutine which itself uses the detokenising routine discussed in chapter 13. When the token is identified, the code which executes it is called before control is finally returned to the *checkend* routine.

```
.comptok                \ identify token
PHA                     \ save token
INY                     \ move index on
JSR addy                \ add Y to contents of r1
PLA                     \ restore token
JSR indirect            \ find token and JMP to
                          its code
.checkend
OPT FNlib("SKIP")       \ move past spaces
CMP #13                 \ is it end of line?
BNE mistake             \ no - there's a mistake!
OPT FNlib("INC1")       \ increment r1
JMP passloop2           \ and do next line
```

Finally *checkend* simply tests to see that the end of line has been reached. If it hasn't, an error is generated; if it has, we move on to compiling the next line in the TACTLESS Language program.

The second pass

The end of a TACTLESS Language program is identified by the &FF (255) byte – when this is identified by the BMI instruction at the beginning of the *passloop2* routine, control is moved on to the routine labelled *nextpass*. Once here, an RTS is compiled at the very end of the code compiled during the first pass. At this time we should only consider this compiled code as dummy in nature since many items of information have not been assigned – as such, it is simply a way of marking space. The RTS is compiled using a subroutine call to *end*. This is a safety net – ideally all

TACTLESS Language programs should have an END as their very last command to make sure that an RTS is compiled at the end of the compiled code – this RTS is compiled just in case a program has an END omitted; at worst, there will be two RTS's!

The vector at *write* now contains the first free byte above the memory needed for the compiled code – this value can now be moved across into the vector *vspace* to assign the actual space allocated for variables. Finally the pass counter is decremented – if this is zero a branch to *compdone* is performed, otherwise a jump to *passloop* is performed to set the second pass of the Compiler into motion:

```
.nextpass
JSR end                 \ compile an RTS - just in
                          case
LDA write               \ get next free byte
                          address and
STA vspace              \ seed this into vspace
                          for use
LDA write+1             \ as variable space
STA vspace+1
DEC pass                \ decrement pass counter
BEQ compdone            \ if 0 then compilation
                          complete
JMP passloop            \ else do second pass
```

Once compilation is complete, details on the file length (including the Library) are displayed and the stand-alone compile code can then be saved. The filename used for the save will have been given as part of the COMPILE command. For example, to compile a TACTLESS Language program to display help information on TACTLESS we could use:

```
COMPILE thelp
```

Once compiled, the code will be saved as a file called *thelp*. The *compdone* machine-code proceeds like this:

```
.compdone
JSR osnewl              \ print a new line
OPT FNlib("PRSTR")      \ print string that
                          follows this
OPT FNequr("Compiled to ")
OPT FNlib("PUSHIND")    \ push value stored at
                          following addr
OPT FNequw(vspace)      \ assemble address
OPT FNlib("PUSHDIR")    \ push following value
OPT FNequw(stackend%)   \ assemble value of
                          stackend%
OPT FNlib("SUBSTRACT")\ subtract numbers on top
                          of stack
OPT FNlib("PRNUM")      \ print number on top of
                          stack
OPT FNlib("PRSTR")      \ print string that
                          follows
OPT FNequr(" bytes")
JSR osnewl              \ print a new line
```

The final section of code that is appended (but not shown here) simply sets up the *osfile* parameter block, pointing the parameter block to the filename and placing the value of *vspace* and *lowest%* into it as the start and end areas to be saved, before jumping to OSFILE with the accumulator holding 0.

Errors Reference has been made to error-handling on several occasions. The *mistake* routine marks entry to the general error-handling routine during compilation and this proceeds as follows:

```
.mistake
OPT FNlib("PRSTR")      \ print following string
OPT FNequr("Mistake")
.comperr
OPT FNlib("PRSTR")      \ print following string
OPT FNequr(" at line ")
LDY #0                  \ clear index
LDA (cur),Y             \ and move current line
STA r2+1                \ into r2
INY                     \ Y=1
```

```
LDA (cur),Y
STA r2
OPT FNlib("PUSH2")        \ push r2 onto stack
OPT FNlib("PRNUM")        \ print number on stack
OPT FNerr("Compilation aborted") \ and finish
```

Using TACTLESS

After all this hard work, now to get the TACTLESS Compiler up and running! First enter the entire listing given as listing 16.1 and save the source on your master disc as "EDSRCE". You will then need to copy the files LIBINFO and LIBCODE on another disc (or you could put it all on one disc as long as there is enough room).

Now RUN the EDSRCE listing (16.1). The screen will clear and the message:

"Reading library info"

will be displayed. At this time the previously created LIBINFO file will be read into the *lib%* and *lib$* arrays as already described. After several seconds a new double message will appear on the screen:

Tactless Editor/Compiler
Calculating code length

There will be a pause as the code is first pseudo-assembled at &D000; then the bottom message will be replaced by another:

Editor/Compiler length : 2327 bytes

Again there will be a pause as the code is reassembled into memory at the correct location. This will take a few moments and finally another message will appear on the screen and the machine code of TACTLESS will be saved. The final message itself will vary depending on what the value of PAGE is in your machine – on my BBC B, PAGE is at &1900 and compiling the code in Mode 7 gives me a final message of:

```
*SAVE TACT 72E9 +917 730A
```

Editor/Compiler saved

after which the > prompt of BASIC reappears.

Note: *Electron owners and readers with tape-based systems should refer to the notes on pages 254 & 255.*

To use TACTLESS all you need do is to make sure that the disc containing TACT and LIBCODE is in the current drive, and type:

```
*TACT
```

Once TACTLESS has loaded it will automatically load in the Library code and then present you with its sign-on message:

Tactless – ready
Bytes used – 0
Bytes free – 20449
#

The bytes-free values may vary – again, this depends on your machine's value of PAGE; the # is the Tactless prompt. Try pressing the RETURN key a few times. The message:

Not direct

will be displayed, indicating the command you just entered was not a direct one – the same will happen if you enter a proper non-direct one – the same will happen if you enter a proper non-direct command such as LOOP, JUMPTO and so on.

The memory information given at the top of the screen can be displayed at any time just by entering the direct command, INFO. These details are also displayed whenever you LOAD a TACTLESS Language program.

Listing 16.2 is a TACTLESS program which demonstrates a few of the program's commands. Enter this and save it as follows:

```
SAVE t.demo
```

It's a good habit to use the *t* directory for TACTLESS programs – if you are using another storage medium which does not allow directories you could just call the file *tdemo*. Now we are ready to compile the program, which we do as follows:

```
COMPILE demo
```

Hey presto! The program will have been compiled and the message:

Compiled to 940 bytes

will be displayed, showing the length of the demo machine code file, before the # prompt reappears. In fact the program itself compiles to just 216 bytes – the remaining bytes are taken up by the Library code.

To test the code simply run it thus:

```
*DEMO
```

The program will count and display numbers from 1 to 99 using a LOOP . . . WHILE structure, issue a beep, and then count back down to 0 using an IF . . . CONTINUE structure.

As you have run the code from within TACTLESS you will return to TACTLESS and its # prompt – the machine-code can be run from anywhere and indeed even on a friend's BBC Micro, even if he doesn't have access to the compiler itself! One final point – you cannot use the OSC command within TACTLESS programs to call other already compiled programs – in such instances incorporate the 'other' program into your current program.

Electron Plus 3

Readers with Electron Plus 3 systems will experience problems. This is because of the high setting to PAGE, typically &1D00 when running under ADFS, coupled with the use of a Mode 6 screen. All is not lost, however. First the following alterations must be made to the program as listed in Listing 16.1:

```
   60 MODE 6
   61 HIMEM=&8000
   62 LOMEM=&6000+3*320
   63 VDU 28,0,2,39,0
  260 edstart%=&6000-edlen%
11030 PRINT t$:ENDPROC
```

Delete the following lines:

270, 11040, 11050

Use EQU in place of the OPT FNequ throughout the program, ie,

```
EQUW replaces OPT FNequw
EQUB  replaces OPT FNequb
EQUS  replaces OPT FNequs
EQUD  replaces OPT FNequd
```

Now save this program to disc. There is still not enough memory for the program to run correctly, so now you will need to compact the program into multi-statement lines, as tightly as you possibly can. After packing a dozen lines or so, save what you have (using a new filename) and RUN the program. When the program runs the screen will have rubbish displayed on it – do not worry. When the machine-code is saved successfully the compacting is complete and the program is now ready to use as described above. If the RUN is aborted, compact the program further.

An alternative method would be to disconnect the Plus 3 and use the tape method to assemble the code, ensuring the Library is assembled above &1D00. Then, reconnect the Plus 3 and transfer the machine code files to disc. Note that a Plus 3 version of the disc is available – see Appendix E.

Tape users

TACTLESS works perfectly with tape. Assemble the programs as already described. To use TACTLESS you will need to ensure that the two files TACT and LIBCODE are stored one after the other. This is best done on a blank cassette. To use, simply position the tape before the TACT file and type:

```
*RUN TACT
```

TACTLESS will be loaded, and this will in turn load the LIBCODE file, after which the # prompt will appear and the Compiler is ready for use. Note that all future compiled programs must be *RUN. Thus in the above example the demo program would be run using:

```
*RUN demo
```

ROM Filing System

The Compiler can be used in conjunction with the ROM Filing System. Using a suitable formatter the TACT and LIBCODE files can be formatted into a sideways RAM image or blown into an EPROM. Once positioned the RFS can be selected and TACTLESS loaded. For further details see Appendix E.

Anatomy

If you are still slightly confused as to exactly what form the compiled code takes, this final excursion inside TACTLESS may just put your mind at rest. The TACTLESS listing 16.2 contains a good selection of the language's commands. At a lower level these compile in a variety of ways – examining a disassembly listing of the compiled code itself will provide a better understanding of the way in which it works.

Figure 16.2 below shows a typical disassembly listing of the compiled code of the DEMO program – note that it only shows the compiled code and not the Library code. If you look at disassemblies of your own code then the actual addresses may vary depending on just where your Library code was originally assembled. In my case the Library code assembled from &2000 to &2307 and the compiled code from &2308 onwards. Typing *INFO DEMO will provide all the details you need. For example, if the output is:

```
$.demo 002000 2308 03AC
```

We can see that the code starts at &2000, the execution address at run-time is &2308, and so this marks the start of the compiled program.

```
Addr  Instruc      Mnemonic    Notes
2308: 20 00 20:    JSR &2000   JSR setstack
230B: 20 7C 20:    JSR &207C   JSR pushdir
230E: 00 00 :                  value=0
2310: 20 9E 20:    JSR &209E   JSR pullind
2313: AC 23 :                  address=&23AC
2315: 20 89 20:    JSR &2089   JSR pushind
2318: AC 23 :                  address=&23AC
231A: 20 7C 20:    JSR &207C   JSR pushdir
231D: 01 00 :                  value=1
231F: 20 03 21:    JSR &2103   JSR add
```

```
2322: 20 9E 20:  JSR  &209E   JSR pullind
2325: AC 23   :               address=&23AC
2327: 20 E2 21:  JSR  21E2    JSR prstr
232A: 43 6F   :               "co
232C: 75 6E   :                  un
232E: 74 3D   :                     t="
2330: 0D      :               return
2331: 20 89 20:  JSR  &2089   JSR pushind
2334: AC 23   :               address=&23AC
2336: 20 4E 22:  JSR  &224E   JSR prnum
2339: 20 E7 FF:  JSR  &FFE7   JSR osnewl
233C: 20 89 20:  JSR  &2089   JSR pushind
233F: AC 23   :               address=&23AC
2340: 20 7C 20:  JSR  &207C   JSR pushdir
2343: 64 00   :               value=100
2346: 20 8C 22:  JSR  &228C   JSR lessthan
2349: 20 84 22:  JSR  &2284   JSR flagtrue
234C: F0 03   :  BEQ  &2351   BEQ &2351
234E: 4C 15 23:  JMP  &2315   JMP next
2351: 20 7C 20:  JSR  &207C   JSR pushdir
2354: 07 00   :               value=7
2356: 20 D7 22:  JSR  &22D7   JSR vdu
2359: 20 89 20:  JSR  &2089   JSR pushind
235C: AC 23   :               address=&23AC
235E: 20 7C 20:  JSR  207C    JSR pushdir
2361: 00 00   :               value=0
2363: 20 C0 22:  JSR  &22C0   JSR notequalto
2366: 20 84 22:  JSR  &2284   JSR flagtrue
2369: D0 03   :  BNE  &236E   BNE &236E
236B: 4C 9C 23:  JMP  &239C   JMP continue
236E: 20 E2 21:  JSR  &21E2   JSR prstr
2371: 43      :               "C
2372: 75 72   :                  ur
2374: 72 65   :                     re
2376: 6E 74 6C:                        nt
2379: 79 3D 0D:                           ly="
237C: 20 89 20:  JSR  &2089   JSR pushind
237F: AC 23   :               address=&23AC
2381: 20 4E 22:  JSR  &224E   JSR prnum
2384: 20 E7 FF:  JSR  &FFE7   JSR osnewl
2387: 20 89 20:  JSR  &2089   JSR pushind
238A: AC 23   :               address=&23AC
238C: 20 7C 20:  JSR  207C    JSR pushdir
238F: 01 00   :               value=1
```

```
2391: 20 1F 21:  JSR  &211F   JSR subtract
2394: 20 9E 20:  JSR  &209E   JSR pullind
2397: AC 23   :               address=&23AC
2399: 4C 59 23:  JMP  &2359   JMP next
239C: 20 E2 21:  JSR  &21E2   JSR prstr
239F: 41 6C 6C:               "All
23A1: 64 6F   :                  do
23A3: 6E 65   :                    ne
23A5: 65 0D   :                      !"
23A7: 20 E7 FF:  JSR  &FFE7   JSR osnewl
23AA: 60      :  RTS          RTS
23AB: 60      :  RTS          RTS
23AC:         :               variable storage
```

Figure 16.2: A disassembly of listing 16.2

A brief study of the table should clear up any lingering doubts you may have had about the way in which a TACTLESS language program is compiled. As always, the very first JSR into the library is to set the stack; and after this comes the program compilation. The first line of the program:

```
10 NOTE This is a demo
```

is ignored by the compiler and does not generate any machine code. The second line, line 20:

```
20 NUMERIC count
```

would have assigned a 5-byte variable space at run-time into which it would place the address of the two bytes to be used by this variable after the first pass. At Line 30,

```
30 count=0
```

we get our first item of compiled code:

```
JSR pushdir
00 00
```

The two bytes after the JSR are pushed on the stack, and these two bytes are the value that the variable is originally set to. This value now has to be assigned to the variable, or more correctly to the address which the compiler is using as that variable:

```
JSR pullind
AC 23
```

The two bytes are pulled from the stack and placed into the two-byte address that follows. We can see from this that the address assigned to *count* during the second pass of the compiler was &23AC.

Line 40 is simply a LOOP command. It does not compile any code, but puts the current address (&231A) onto the loop stack.

Line 50 increments count by 1,

```
50 count=count+1
```

Again, the increment value, 1 in this case, is compiled after a *JSR pushind* to ensure that the 1 is put on the stack. This is followed by a *JSR add* which places the sum of the two numbers on the stack on the top of the stack before a *JSR pullind* stores this result at the address which follows it. The code compiled by line 50 is therefore

```
JSR pushdir
01 00
JSR add
JSR pullind
AC 23
```

Line 60,

```
60 PRINT "Count=";
```

is simply compiled and involves a *JSR prnst* followed by the ASCII string itself. The &0D RETURN character at the end does not force a newline, as OSWRCH is used as the output routine. Line 70 prints the current contents of the variable. To do this, the variable's value is extracted from &23AC and printed using the *prnum* routine. Line 70 therefore compiles the following code:

```
JSR pushind
AC 23
JSR prnum
JSR osnewl
```

Because the line does not end in a semicolon a new line is also printed.

Line 80 is a bit more complex as the code it compiles must evaluate, test and branch as needed.

```
80 WHILE count < 100
```

The first step is to push the contents of *count* on the stack, which is done using a *JSR pushind*; a *JSR pushdir* then pushes the value (100) it is to be compared to on the stack, thus:

```
JSR pushdir
AC 23
JSR pushdir
64 00
```

The two values on top of the stack are then tested against each other and if the first value is not less than the second the *JSR flagtrue* will set the zero flag in the status register so that the BEQ will cause the loop to be left. On the other hand, if the result is false then a JMP will occur to the address marked earlier by LOOP – obviously, this address was pulled from the loop stack at compile time.

```
JSR lessthan
JSR flagtrue
BEQ over
JMP next
.over
```

The VDU 7 command of line 90 is simple enough. A *JSR pushdir* pushes the VDU code following it on the stack, which is then issued by the *JSR vdu* thus:

```
JSR pushdir
07 00
JSR vdu
```

Line 100 is a label and its address will be held in the variable space during the compilation. It will be extracted during the second pass when it is needed by

JUMPTO.

Line 110 involves another conditional test:

```
110 IF count<>0
```

This compiles in a not too dissimilar way to the WHILE code above, except that the conditional JSR this time is to *notequal*. The code compiled is:

```
JSR pushind
AC 23
JSR pushdir
00 00
JSR notequal
JSR flagtrue
BEQ over2
JMP continue
.over2
```

The *JSR prstr* is used to print the string defined in line 120 and this is followed by the number currently in &23AC, as required by line 130.

The subtraction in line 140 is performed in exactly the same way as the addition in line 50 but the *JSR add* is replaced with a *JSR subtract*. Not surprisingly, a *JMP next* is compiled by the JUMPTO in line 150, the previously stored label address being extracted from the variable space during compilation. The CONTINUE of line 160 is used by the IF command of line 110 during the second pass of the compiler. The 'Alldone' string is compiled after a *JSR prstr* (line 170) and, finally, the END of line 180 compiles an RTS, followed by the safety net RTS.

You might like to try writing a few of your own programs in the TACTLESS language and then disassemble them in a similar way. If you do, then program 16.3 may be of some help – it reads in the LIBINFO file and prints the names and execution addresses of all the library routines. Disassembling the programs in this way will certainly help you understand the inner workings of the TACTLESS compiler.

Extending TACTLESS

You will want to extend your TACTLESS Compiler by adding more commands and facilities. This is quite straightforward to do, but it is wise to start with simple commands and move on to more complex ones as your understanding develops. Ideally try to base your commands on those that already exist. For example, we could add our old friend the BEEP command. In essence this is simply a VDU 7 and could be matched with the SCREEN command which also uses the VDU driver. Obviously there are a number of stages involved when adding a command, and in summary they are:

- add the code to the Library along with LIBINFO details
- extend the *lib$* and *lib%* DIM statement
- add intermediate code to the TACTLESS source program
- add the command, token and label to the token table
- reassemble everything
- test it!

Extending the two source programs indefinitely is not possible because of memory restrictions. Appendix A contains details on writing long assembly language programs and this should be of great help. An alternative approach would be to split the TACTLESS source so that the Editor and Compiler become two separate programs. The more adventurous among you might even contemplate configuring the whole suite to run from sideways RAM which would also have the benefit of leaving the entire user RAM free for TACTLESS language programs . . .

Listing 16.1

```
10 REM Tactless Editor/Compiler
20 REM Bruce Smith/Dave Acton
30 REM BBC B/B+/Electron/Master
40 REM (C) Bruce Smith 1986
50 :
60 MODE 7
70 ON ERROR PROCreport
80 DIM lib$(40),lib%(40)
90 PROCcentre(2,"Reading library info")
100 in%=OPENUP"LIBINFO"
```

```
11Ø INPUT#in%,stack%,stackend%
12Ø INPUT#in%,liblen%
13Ø lowest%=stackend%+liblen%
14Ø k%=Ø
15Ø REPEAT
16Ø INPUT#in%,lib$(k%),lib%(k%)
17Ø k%=k%+1
18Ø UNTIL lib$(k%-1)="@"
19Ø CLOSE#in%
2ØØ PROCinit
21Ø PROCcentre(2,"Tactless Editor/Compiler")
22Ø PROCcentre(4,"Calculating code length")
23Ø edstart%=&DØØØ
24Ø PROCass(&DØØØ)
25Ø edlen%=P%-&DØØØ
26Ø edstart%=HIMEM-edlen%
27Ø HIMEM=edstart%
28Ø PROCcentre(4,"Editor/compiler length
    : "+STR$edlen%+" bytes")
29Ø PROCass(edstart%)
3ØØ PROCsavecode
31Ø END
32Ø :
33Ø DEF PROCinit
34Ø DIM os% 4Ø
35Ø input%=&6ØØ:osf%=&7ØØ
36Ø opst%=&5ØØ:loopst%=&54Ø
37Ø ifst%=&58Ø
38Ø brkv=&2Ø2
39Ø oscli=&FFF7:oswrch=&FFEE
4ØØ osbyte=&FFF4:osword=&FFF1
41Ø osnewl=&FFE7:osasci=&FFE3
42Ø osfile=&FFDD
43Ø page=&7Ø
44Ø himem=&72
45Ø write=&74
46Ø vspace=&76
47Ø vcomp=&78
48Ø jump=&7A
49Ø stp=&7C
5ØØ varpt=&7E
51Ø loopstp=&8Ø
52Ø ifstp=&81
```

```
530 opstp=&82
540 varflag=&83
550 cur=&84
560 count=&86
570 temp=&87
580 len=&88
590 pass=&89
600 tokflag=&8A
610 r1=&20:r2=&22:r3=&24
620 ENDPROC
630 :
640 DEF PROCass(editor%)
650 FOR pass%=0 TO 2 STEP 2
660 P%=editor%
670 [OPT pass%
680 .error
690 LDA #126
700 LDX #0
710 LDY #0
720 JSR osbyte
730 JSR osnewl
740 LDY #1
750 .errloop
760 LDA (&FD),Y
770 JSR oswrch
780 INY
790 CMP #0
800 BNE errloop
810 JSR osnewl
820 OPT FNlib("SETSTACK")
830 JMP tactloop
840 :
850 .tact
860 LDA #error MOD 256
870 STA brkv
880 LDA #error DIV 256
890 STA brkv+1
900 LDX #loadlib MOD 256
910 LDY #loadlib DIV 256
920 JSR oscli
930 JSR new
940 OPT FNlib("SETSTACK")
950 OPT FNlib("PRSTR")
```

```
 96Ø OPT FNequb(22)
 97Ø OPT FNequb(7)
 98Ø OPT FNequs("Tactless - ready")
 99Ø OPT FNequb(1Ø)
1ØØØ OPT FNequb(13)
1Ø1Ø JSR osnewl
1Ø2Ø JSR info
1Ø3Ø .tactloop
1Ø4Ø LDA #35
1Ø5Ø JSR oswrch
1Ø6Ø OPT FNlib("INPUTBUFF")
1Ø7Ø CMP #42
1Ø8Ø BNE notstar
1Ø9Ø LDX r1
11ØØ LDY r1+1
111Ø JSR oscli
112Ø JMP tactloop
113Ø .notstar
114Ø OPT FNlib("NUM")
115Ø BCC line
116Ø JSR tokenise
117Ø LDA input%
118Ø CMP #&F9
119Ø BCC notdirect
12ØØ JSR indirect
121Ø JMP tactloop
122Ø .notdirect
123Ø OPT FNerr("Not direct")
124Ø .line
125Ø OPT FNlib("CONV")
126Ø OPT FNlib("SKIP")
127Ø JSR tokenise
128Ø OPT FNlib("PULL3")
129Ø OPT FNlib("PUSH3")
13ØØ JSR deleteline
131Ø LDA input%
132Ø CMP #13
133Ø BEQ tactloop
134Ø LDA page
135Ø STA r2
136Ø LDA page+1
137Ø STA r2+1
138Ø .insertloop
```

```
139Ø LDY #Ø
14ØØ LDA (r2),Y
141Ø BMI insertthere
142Ø LDA r3+1
143Ø CMP (r2),Y
144Ø BCC insertthere
145Ø BNE lessline
146Ø INY
147Ø LDA r3
148Ø CMP (r2),Y
149Ø BCC insertthere
15ØØ .lessline
151Ø JSR lineskip
152Ø JMP insertloop
153Ø .insertthere
154Ø LDA page
155Ø STA r1
156Ø SEC
157Ø SBC len
158Ø STA page
159Ø STA r3
16ØØ LDA page+1
161Ø STA r1+1
162Ø SBC #Ø
163Ø STA page+1
164Ø STA r3+1
165Ø .movedown
166Ø LDA r1
167Ø CMP r2
168Ø BNE notmoveend
169Ø LDA r1+1
17ØØ CMP r2+1
171Ø BEQ moveend
172Ø .notmoveend
173Ø LDY #Ø
174Ø LDA (r1),Y
175Ø STA (r3),Y
176Ø OPT FNlib("INC1")
177Ø OPT FNlib("INC3")
178Ø JMP movedown
179Ø .moveend
18ØØ OPT FNlib("PULL2")
181Ø LDA r2+1
```

```
182Ø STA (r3),Y
183Ø INY
184Ø LDA r2
185Ø STA (r3),Y
186Ø INY
187Ø LDA len
188Ø STA (r3),Y
189Ø LDX #255
19ØØ .copyloop
191Ø INX
192Ø INY
193Ø LDA input%,X
194Ø STA (r3),Y
195Ø CMP #13
196Ø BNE copyloop
197Ø JMP tactloop
198Ø :
199Ø .indirect
2ØØØ JSR findtok
2Ø1Ø INY
2Ø2Ø LDA (r3),Y
2Ø3Ø STA jump
2Ø4Ø INY
2Ø5Ø LDA (r3),Y
2Ø6Ø STA jump+1
2Ø7Ø JMP (jump)
2Ø8Ø :
2Ø9Ø .new
21ØØ LDA #(edstart%-3) MOD 256
211Ø STA page
212Ø STA himem
213Ø LDA #(edstart%-3) DIV 256
214Ø STA page+1
215Ø STA himem+1
216Ø LDA #255
217Ø STA edstart%-3
218Ø RTS
219Ø :
22ØØ .list
221Ø LDA #(input%+1) MOD 256
222Ø STA r1
223Ø LDA #(input%+1) DIV 256
224Ø STA r1+1
```

```
225Ø OPT FNlib("SKIP")
226Ø OPT FNlib("NUM")
227Ø BCS listall
228Ø OPT FNlib("CONV")
229Ø OPT FNlib("PULL3")
23ØØ JSR findline
231Ø BCS nosuchline
232Ø JMP listline
233Ø .nosuchline
234Ø OPT FNerr("No such line")
235Ø .listall
236Ø LDA page
237Ø STA r2
238Ø LDA page+1
239Ø STA r2+1
24ØØ .listloop
241Ø LDY #Ø
242Ø LDA (r2),Y
243Ø BMI endlist
244Ø JSR listline
245Ø JMP listloop
246Ø .endlist
247Ø RTS
248Ø :
249Ø .listline
25ØØ LDY #Ø
251Ø LDA (r2),Y
252Ø STA r1+1
253Ø INY
254Ø LDA (r2),Y
255Ø STA r1
256Ø OPT FNlib("PUSH1")
257Ø OPT FNlib("PRNUM")
258Ø OPT FNlib("INC2")
259Ø OPT FNlib("INC2")
26ØØ .text
261Ø OPT FNlib("INC2")
262Ø LDY #Ø
263Ø LDA (r2),Y
264Ø BPL ord
265Ø JSR findtok
266Ø LDY #Ø
267Ø .disptok
```

```
268Ø LDA (r3),Y
269Ø BMI text
27ØØ JSR osasci
271Ø INY
272Ø BNE disptok
273Ø .ord
274Ø JSR osasci
275Ø CMP #13
276Ø BNE text
277Ø OPT FNlib("INC2")
278Ø RTS
279Ø :
28ØØ .clearosf
281Ø LDA #Ø
282Ø LDX #17
283Ø .clearloop
284Ø STA osf%,X
285Ø DEX
286Ø BPL clearloop
287Ø RTS
288Ø :
289Ø .save
29ØØ LDA page
291Ø STA edstart%-2
292Ø LDA page+1
293Ø STA edstart%-1
294Ø JSR clearosf
295Ø LDA page
296Ø STA osf%+1Ø
297Ø STA osf%+2
298Ø LDA page+1
299Ø STA osf%+11
3ØØØ STA osf%+3
3Ø1Ø LDA #edstart% MOD 256
3Ø2Ø STA osf%+14
3Ø3Ø LDA #edstart% DIV 256
3Ø4Ø STA osf%+15
3Ø5Ø LDA #Ø
3Ø6Ø .file
3Ø7Ø LDX #(input%+1) MOD 256
3Ø8Ø STX osf%
3Ø9Ø LDX #(input%+1) DIV 256
31ØØ STX osf%+1
```

```
3110 LDX #osf% MOD 256
3120 LDY #osf% DIV 256
3130 JMP osfile
3140 :
3150 .load
3160 JSR clearosf
3170 INC osf%+6
3180 LDA #&FF
3190 JSR file
3200 LDA edstart%-2
3210 STA page
3220 LDA edstart%-1
3230 STA page+1
3240 :
3250 .info
3260 OPT FNlib("PRSTR")
3270 OPT FNequr("Bytes used - "
3280 OPT FNlib("PUSHIND")
3290 OPT FNequw(himem)
3300 OPT FNlib("PUSHIND")
3310 OPT FNequw(page)
3320 OPT FNlib("SUBTRACT")
3330 OPT FNlib("PRNUM")
3340 JSR osnewl
3350 OPT FNlib("PRSTR")
3360 OPT FNequr("Bytes free - "
3370 OPT FNlib("PUSHIND")
3380 OPT FNequw(page)
3390 OPT FNlib("PUSHDIR")
3400 OPT FNequw(lowest%)
3410 OPT FNlib("SUBTRACT")
3420 OPT FNlib("PRNUM")
3430 JMP osnewl
3440 :
3450 .vchar
3460 CMP #97
3470 BCC notvchar
3480 CMP #122
3490 RTS
3500 .notvchar
3510 SEC
3520 RTS
3530 :
```

```
354Ø .tokenise
355Ø LDY #Ø
356Ø LDA (r1),Y
357Ø EOR #42
358Ø STA tokflag
359Ø LDA #(input%-1) MOD 256
36ØØ STA r2
361Ø LDA #(input%-1) DIV 256
362Ø STA r2+1
363Ø LDA #3
364Ø STA len
365Ø .tokenloop
366Ø OPT FNlib("INC2")
367Ø INC len
368Ø JSR findkey
369Ø BCS nottok
37ØØ LDA tokflag
371Ø BEQ nottok
372Ø JSR addy
373Ø LDA (r3),Y
374Ø LDY #Ø
375Ø STA (r2),Y
376Ø JMP tokenloop
377Ø .nottok
378Ø LDY #Ø
379Ø LDA (r1),Y
38ØØ STA (r2),Y
381Ø PHA
382Ø OPT FNlib("INC1")
383Ø PLA
384Ø CMP #13
385Ø BNE tokenloop
386Ø RTS
387Ø :
388Ø .findkey
389Ø LDA #vocab% MOD 256
39ØØ STA r3
391Ø LDA #vocab% DIV 256
392Ø STA r3+1
393Ø .findloop
394Ø LDY #Ø
395Ø .samekey
396Ø LDA (r3),Y
```

```
397Ø BMI endkey
398Ø CMP (r1),Y
399Ø BNE diffkey
4ØØØ INY
4Ø1Ø BNE samekey
4Ø2Ø .endkey
4Ø3Ø CMP #255
4Ø4Ø RTS
4Ø5Ø .notend
4Ø6Ø INY
4Ø7Ø .diffkey
4Ø8Ø LDA (r3),Y
4Ø9Ø BPL notend
41ØØ TYA
411Ø CLC
412Ø ADC #3
413Ø ADC r3
414Ø STA r3
415Ø LDA r3+1
416Ø ADC #Ø
417Ø STA r3+1
418Ø JMP findloop
419Ø :
42ØØ .findtok
421Ø TAX
422Ø LDA #vocab% MOD 256
423Ø STA r3
424Ø LDA #vocab% DIV 256
425Ø STA r3+1
426Ø .ftok2
427Ø LDY #255
428Ø .ftok3
429Ø INY
43ØØ LDA (r3),Y
431Ø BPL ftok3
432Ø CMP #255
433Ø BEQ tokout
434Ø TXA
435Ø CMP (r3),Y
436Ø BEQ tokout
437Ø TYA
438Ø CLC
439Ø ADC #3
```

```
44ØØ ADC r3
441Ø STA r3
442Ø LDA r3+1
443Ø ADC #Ø
444Ø STA r3+1
445Ø JMP ftok2
446Ø .tokout
447Ø RTS
448Ø :
449Ø .findline
45ØØ LDA page
451Ø STA r2
452Ø LDA page+1
453Ø STA r2+1
454Ø .findline2
455Ø LDY #Ø
456Ø LDA (r2),Y
457Ø BMI noline
458Ø CMP r3+1
459Ø BNE findline3
46ØØ INY
461Ø LDA (r2),Y
462Ø CMP r3
463Ø BNE findline3
464Ø CLC
465Ø RTS
466Ø .findline3
467Ø JSR lineskip
468Ø JMP findline2
469Ø .noline
47ØØ SEC
471Ø RTS
472Ø :
473Ø .deleteline
474Ø JSR findline
475Ø BCS notthere
476Ø LDY #2
477Ø LDA (r2),Y
478Ø STA temp
479Ø .deleteloop
48ØØ LDA r2+1
481Ø CMP page+1
482Ø BNE shift
```

```
483Ø LDA r2
484Ø CMP page
485Ø BNE shift
486Ø CLC
487Ø ADC temp
488Ø STA page
489Ø LDA page+1
49ØØ ADC #Ø
491Ø STA page+1
492Ø .notthere
493Ø RTS
494Ø .shift
495Ø OPT FNlib("DEC2")
496Ø LDY #Ø
497Ø LDA (r2),Y
498Ø LDY temp
499Ø STA (r2),Y
5ØØØ JMP deleteloop
5Ø1Ø :
5Ø2Ø .comp
5Ø3Ø LDA #2
5Ø4Ø STA pass
5Ø5Ø LDA page
5Ø6Ø STA varpt
5Ø7Ø LDA page+1
5Ø8Ø STA varpt+1
5Ø9Ø LDA #lowest% MOD 256
51ØØ STA vspace
511Ø LDA #lowest% DIV 256
512Ø STA vspace+1
513Ø .passloop
514Ø LDA #Ø
515Ø STA opstp
516Ø STA loopstp
517Ø STA ifstp
518Ø LDA page
519Ø STA r1
52ØØ LDA page+1
521Ø STA r1+1
522Ø LDA #lowest% MOD 256
523Ø STA write
524Ø LDA #lowest% DIV 256
525Ø STA write+1
```

```
526Ø OPT FNasslib("SETSTACK")
527Ø .passloop2
528Ø LDA r1
529Ø STA cur
53ØØ LDA r1+1
531Ø STA cur+1
532Ø LDY #Ø
533Ø LDA (r1),Y
534Ø BMI nextpass
535Ø LDY #3
536Ø LDA (r1),Y
537Ø BMI comptok
538Ø JSR vchar
539Ø BCC compvar
54ØØ CMP #46
541Ø BNE compstar
542Ø INY
543Ø JSR addy
544Ø LDA #255
545Ø JSR var
546Ø JMP checkend
547Ø .compvar
548Ø JSR addy
549Ø JSR set
55ØØ JMP checkend
551Ø .compstar
552Ø CMP #42
553Ø BNE mistake
554Ø JSR addy
555Ø OPT FNasslib("OSC")
556Ø .compstarloop
557Ø OPT FNlib("INC1")
558Ø LDY #Ø
559Ø LDA (r1),Y
56ØØ JSR compbyte
561Ø CMP #13
562Ø BNE compstarloop
563Ø BEQ checkend
564Ø .comptok
565Ø PHA
566Ø INY
567Ø JSR addy
568Ø PLA
```

```
569Ø JSR indirect
57ØØ .checkend
571Ø OPT FNlib("SKIP")
572Ø CMP #13
573Ø BNE mistake
574Ø OPT FNlib("INC1")
575Ø JMP passloop2
576Ø .nextpass
577Ø JSR end
578Ø LDA write
579Ø STA vspace
58ØØ LDA write+1
581Ø STA vspace+1
582Ø DEC pass
583Ø BEQ compdone
584Ø JMP passloop
585Ø .mistake
586Ø OPT FNlib("PRSTR")
587Ø OPT FNequr("Mistake")
588Ø .comperr
589Ø OPT FNlib("PRSTR")
59ØØ OPT FNequr(" at line ")
591Ø LDY #Ø
592Ø LDA (cur),Y
593Ø STA r2+1
594Ø INY
595Ø LDA (cur),Y
596Ø STA r2
597Ø OPT FNlib("PUSH2")
598Ø OPT FNlib("PRNUM")
599Ø OPT FNerr("Compilation aborted")
6ØØØ .compdone
6Ø1Ø JSR osnewl
6Ø2Ø OPT FNlib("PRSTR")
6Ø3Ø OPT FNequr("Compiled to ")
6Ø4Ø OPT FNlib("PUSHIND")
6Ø5Ø OPT FNequw(vspace)
6Ø6Ø OPT FNlib("PUSHDIR")
6Ø7Ø OPT FNequw(stackend%)
6Ø8Ø OPT FNlib("SUBTRACT")
6Ø9Ø OPT FNlib("PRNUM")
61ØØ OPT FNlib("PRSTR")
611Ø OPT FNequr(" bytes")
```

```
612Ø JSR osnewl
613Ø JSR clearosf
614Ø LDA #stackend% MOD 256
615Ø STA osf%+1Ø
616Ø STA osf%+2
617Ø LDA #stackend% DIV 256
618Ø STA osf%+11
619Ø STA osf%+3
62ØØ LDA vspace
621Ø STA osf%+14
622Ø LDA vspace+1
623Ø STA osf%+15
624Ø LDA #lowest% MOD 256
625Ø STA osf%+6
626Ø LDA #lowest% DIV 256
627Ø STA osf%+7
628Ø LDA #Ø
629Ø JMP file
63ØØ :
631Ø .comp2
632Ø LDA r2
633Ø JSR compbyte
634Ø LDA r2+1
635Ø .compbyte
636Ø LDY #Ø
637Ø STA (write),Y
638Ø INC write
639Ø BNE cb2
64ØØ INC write+1
641Ø .cb2
642Ø RTS
643Ø :
644Ø .sta
645Ø LDA #&8D
646Ø BNE compbyte
647Ø :
648Ø .rpn
649Ø OPT FNlib("SKIP")
65ØØ CMP #13
651Ø BEQ clearops
652Ø OPT FNlib("NUM")
653Ø BCS tryvar
654Ø OPT FNlib("CONV")
```

```
655Ø OPT FNlib("PULL2")
656Ø OPT FNasslib("PUSHDIR")
657Ø JSR comp2
658Ø JMP rpn
659Ø .tryvar
66ØØ JSR vchar
661Ø BCS tryop
662Ø LDA #Ø
663Ø JSR var
664Ø OPT FNasslib("PUSHIND")
665Ø JSR comp2
666Ø JMP rpn
667Ø .tryop
668Ø JSR findop
669Ø BCC assop
67ØØ .clearops
671Ø LDA opstp
672Ø BEQ rpnout
673Ø DEC opstp
674Ø JSR asssub
675Ø JMP clearops
676Ø .rpnout
677Ø RTS
678Ø .lowerpri
679Ø DEC opstp
68ØØ JSR asssub
681Ø .assop
682Ø LDY #1
683Ø LDX opstp
684Ø BEQ pushop
685Ø LDA opst%-1,X
686Ø CMP (r3),Y
687Ø BCS lowerpri
688Ø .pushop
689Ø LDA (r3),Y
69ØØ STA opst%,X
691Ø INC opstp
692Ø INY
693Ø LDA (r3),Y
694Ø STA r2
695Ø INY
696Ø LDA (r3),Y
697Ø STA r2+1
```

```
698Ø OPT FNlib("PUSH2")
699Ø OPT FNlib("INC1")
7ØØØ JMP rpn
7Ø1Ø :
7Ø2Ø .findop
7Ø3Ø LDA #ops% MOD 256
7Ø4Ø STA r3
7Ø5Ø LDA #ops% DIV 256
7Ø6Ø STA r3+1
7Ø7Ø .findoploop
7Ø8Ø LDY #Ø
7Ø9Ø LDA (r3),Y
71ØØ CMP #255
711Ø BNE opcomp
712Ø RTS
713Ø .opcomp
714Ø CMP (r1),Y
715Ø CLC
716Ø BNE nextop
717Ø RTS
718Ø .nextop
719Ø LDA r3
72ØØ ADC #4
721Ø STA r3
722Ø LDA r3+1
723Ø ADC #Ø
724Ø STA r3+1
725Ø JMP findoploop
726Ø :
727Ø .var
728Ø STA varflag
729Ø OPT FNlib("SKIP")
73ØØ LDA page
731Ø STA r3
732Ø LDA page+1
733Ø STA r3+1
734Ø .varloop
735Ø LDA r3
736Ø CMP varpt
737Ø BNE searchforvar
738Ø LDA r3+1
739Ø CMP varpt+1
74ØØ BEQ notfoundbefore
```

```
741Ø .searchforvar
742Ø LDA r3
743Ø SEC
744Ø SBC #5
745Ø STA r3
746Ø LDA r3+1
747Ø SBC #Ø
748Ø STA r3+1
749Ø LDY #Ø
75ØØ LDA (r3),Y
751Ø STA vcomp
752Ø INY
753Ø LDA (r3),Y
754Ø STA vcomp+1
755Ø DEY
756Ø .varcomp
757Ø LDA (vcomp),Y
758Ø JSR vchar
759Ø BCS foundbefore
76ØØ CMP (r1),Y
761Ø BNE varloop
762Ø INY
763Ø BNE varcomp
764Ø .foundbefore
765Ø LDA varflag
766Ø BNE assign
767Ø LDY #2
768Ø LDA (r3),Y
769Ø STA r2
77ØØ INY
771Ø LDA (r3),Y
772Ø STA r2+1
773Ø JMP skipvar
774Ø .notfoundbefore
775Ø LDA pass
776Ø CMP #1
777Ø BNE firstpass
778Ø OPT FNlib("PRSTR")
779Ø OPT FNequr("No such variable/label")
78ØØ JMP comperr
781Ø .firstpass
782Ø LDA varflag
783Ø BEQ skipvar
```

```
784Ø LDA varpt
785Ø SEC
786Ø SBC #5
787Ø STA varpt
788Ø STA r3
789Ø LDA varpt+1
79ØØ SBC #Ø
791Ø STA varpt+1
792Ø STA r3+1
793Ø LDY #Ø
794Ø LDA r1
795Ø STA (r3),Y
796Ø INY
797Ø LDA r1+1
798Ø STA (r3),Y
799Ø .assign
8ØØØ LDY #2
8Ø1Ø LDA varflag
8Ø2Ø BMI labelonly
8Ø3Ø LDA vspace
8Ø4Ø STA (r3),Y
8Ø5Ø STA r2
8Ø6Ø INY
8Ø7Ø LDA vspace+1
8Ø8Ø STA (r3),Y
8Ø9Ø STA r2+1
81ØØ LDA vspace
811Ø CLC
812Ø ADC #2
813Ø STA vspace
814Ø LDA vspace+1
815Ø ADC #Ø
816Ø STA vspace+1
817Ø JMP skipvar
818Ø .labelonly
819Ø LDA write
82ØØ STA (r3),Y
821Ø STA r2
822Ø INY
823Ø LDA write+1
824Ø STA (r3),Y
825Ø STA r2+1
826Ø .skipvar
```

```
827Ø OPT FNlib("INC1")
828Ø LDY #Ø
829Ø LDA (r1),Y
83ØØ JSR vchar
831Ø BCC skipvar
832Ø RTS
833Ø :
834Ø .numer
835Ø LDA #1
836Ø JMP var
837Ø :
838Ø .set
839Ø LDA #Ø
84ØØ JSR var
841Ø OPT FNlib("PUSH2")
842Ø OPT FNlib("SKIP")
843Ø CMP #61
844Ø BEQ equals
845Ø JMP mistake
846Ø .equals
847Ø OPT FNlib("INC1")
848Ø JSR rpn
849Ø OPT FNlib("PULL2")
85ØØ OPT FNasslib("PULLIND")
851Ø JMP comp2
852Ø :
853Ø .loop
854Ø LDX loopstp
855Ø LDA write
856Ø STA loopst%,X
857Ø LDA write+1
858Ø STA loopst%+1,X
859Ø INC loopstp
86ØØ INC loopstp
861Ø RTS
862Ø :
863Ø .while
864Ø JSR rpn
865Ø OPT FNasslib("FLAGTRUE")
866Ø LDA #&FØ
867Ø JSR compbyte
868Ø LDA #3
869Ø JSR compbyte
```

```
87ØØ LDA #&4C
871Ø JSR compbyte
872Ø DEC loopstp
873Ø DEC loopstp
874Ø LDX loopstp
875Ø LDA loopst%,X
876Ø JSR compbyte
877Ø LDA loopst%+1,X
878Ø JMP compbyte
879Ø :
88ØØ .note
881Ø LDY #255
882Ø .noteloop
883Ø INY
884Ø LDA (r1),Y
885Ø CMP #13
886Ø BNE noteloop
887Ø .addy
888Ø TYA
889Ø CLC
89ØØ ADC r1
891Ø STA r1
892Ø LDA r1+1
893Ø ADC #Ø
894Ø STA r1+1
895Ø RTS
896Ø :
897Ø .print
898Ø OPT FNlib("SKIP")
899Ø CMP #34
9ØØØ BEQ string
9Ø1Ø CMP #13
9Ø2Ø BEQ printret
9Ø3Ø CMP #59
9Ø4Ø BNE printv
9Ø5Ø OPT FNlib("INC1")
9Ø6Ø LDY #Ø
9Ø7Ø LDA (r1),Y
9Ø8Ø CMP #13
9Ø9Ø BNE print
91ØØ RTS
911Ø .printv
912Ø JSR vchar
```

```
913Ø BCC okv
914Ø OPT FNlib("NUM")
915Ø BCC okv
916Ø JMP mistake
917Ø .okv
918Ø JSR rpn
919Ø OPT FNasslib("PRNUM")
92ØØ JMP print
921Ø .string
922Ø OPT FNasslib("PRSTR")
923Ø .stringloop
924Ø OPT FNlib("INC1")
925Ø LDY #Ø
926Ø LDA (r1),Y
927Ø CMP #34
928Ø BEQ stringout
929Ø JSR compbyte
93ØØ CMP #13
931Ø BNE stringloop
932Ø OPT FNerr("Missing """)
933Ø .stringout
934Ø LDA #13
935Ø JSR compbyte
936Ø OPT FNlib("INC1")
937Ø JMP print
938Ø .printret
939Ø OPT FNlib("PUSHDIR")
94ØØ OPT FNequw(osnewl)
941Ø .asssub
942Ø LDA #&2Ø
943Ø JSR compbyte
944Ø OPT FNlib("PULL2")
945Ø JMP comp2
946Ø :
947Ø .callsub
948Ø LDA #Ø
949Ø JSR var
95ØØ OPT FNlib("PUSH2")
951Ø JMP asssub
952Ø :
953Ø .vdu
954Ø JSR rpn
955Ø OPT FNasslib("VDU")
```

```
956Ø OPT FNlib("SKIP")
957Ø CMP #44
958Ø BNE vduout
959Ø OPT FNlib("INC1")
96ØØ JMP vdu
961Ø .vduout
962Ø RTS
963Ø :
964Ø .if
965Ø JSR rpn
966Ø OPT FNasslib("FLAGTRUE")
967Ø LDA #&DØ
968Ø JSR compbyte
969Ø LDA #3
97ØØ JSR compbyte
971Ø LDA #&4C
972Ø JSR compbyte
973Ø LDX ifstp
974Ø LDA write
975Ø STA ifst%,X
976Ø LDA write+1
977Ø STA ifst%+1,X
978Ø INC ifstp
979Ø INC ifstp
98ØØ JSR compbyte
981Ø JMP compbyte
982Ø :
983Ø .cont
984Ø DEC ifstp
985Ø DEC ifstp
986Ø LDX ifstp
987Ø LDA ifst%,X
988Ø STA r3
989Ø LDA ifst%+1,X
99ØØ STA r3+1
991Ø LDY #Ø
992Ø LDA write
993Ø STA (r3),Y
994Ø INY
995Ø LDA write+1
996Ø STA (r3),Y
997Ø RTS
998Ø :
```

```
 999Ø .end
1ØØØØ LDA #&6Ø
1ØØ1Ø JMP compbyte
1ØØ2Ø :
1ØØ3Ø .jumpto
1ØØ4Ø LDA #Ø
1ØØ5Ø JSR var
1ØØ6Ø LDA #&4C
1ØØ7Ø JSR compbyte
1ØØ8Ø JMP comp2
1ØØ9Ø :
1Ø1ØØ .input
1Ø11Ø LDA #Ø
1Ø12Ø JSR var
1Ø13Ø OPT FNasslib("INPUTBUFF")
1Ø14Ø OPT FNasslib("CONV")
1Ø15Ø OPT FNasslib("PULLIND")
1Ø16Ø JMP comp2
1Ø17Ø :
1Ø18Ø .screen
1Ø19Ø JSR rpn
1Ø2ØØ OPT FNasslib("SCREEN")
1Ø21Ø RTS
1Ø22Ø :
1Ø23Ø .lineskip
1Ø24Ø LDY #2
1Ø25Ø LDA (r2),Y
1Ø26Ø CLC
1Ø27Ø ADC r2
1Ø28Ø STA r2
1Ø29Ø LDA r2+1
1Ø3ØØ ADC #Ø
1Ø31Ø STA r2+1
1Ø32Ø RTS
1Ø33Ø :
1Ø34Ø .loadlib
1Ø35Ø OPT FNequr("L.LIBCODE")
1Ø36Ø :
1Ø37Ø .vocab%
1Ø38Ø OPT FNtok("IF",&8Ø,if)
1Ø39Ø OPT FNtok("CONTINUE",&81,cont)
1Ø4ØØ OPT FNtok("WHILE",&82,while)
1Ø41Ø OPT FNtok("NUMERIC",&83,numer)
```

```
1Ø42Ø OPT FNtok("LOOP",&84,loop)
1Ø43Ø OPT FNtok("PRINT",&85,print)
1Ø44Ø OPT FNtok("NOTE",&86,note)
1Ø45Ø OPT FNtok("INPUT",&87,input)
1Ø46Ø OPT FNtok("END",&88,end)
1Ø47Ø OPT FNtok("SCREEN",&89,screen)
1Ø48Ø OPT FNtok("VDU",&8A,vdu)
1Ø49Ø OPT FNtok("CALLSUB",&8B,callsub)
1Ø5ØØ OPT FNtok("JUMPTO",&8C,jumpto)
1Ø51Ø OPT FNtok("AND",&AØ,notdirect)
1Ø52Ø OPT FNtok("OR",&A1,notdirect)
1Ø53Ø OPT FNtok("DIV",&A2,notdirect)
1Ø54Ø OPT FNtok("MOD",&A3,notdirect)
1Ø55Ø OPT FNtok("<>",&A4,notdirect)
1Ø56Ø OPT FNtok("LIST",&F9,list)
1Ø57Ø OPT FNtok("INFO",&FA,info)
1Ø58Ø OPT FNtok("COMPILE",&FB,comp)
1Ø59Ø OPT FNtok("SAVE",&FC,save)
1Ø6ØØ OPT FNtok("LOAD",&FD,load)
1Ø61Ø OPT FNtok("NEW",&FE,new)
1Ø62Ø OPT FNequb(&FF)
1Ø63Ø :
1Ø64Ø .ops%
1Ø65Ø OPT FNop("*",8,"MULT")
1Ø66Ø OPT FNop(CHR$&A2,8,"DIV")
1Ø67Ø OPT FNop(CHR$&A3,8,"MOD")
1Ø68Ø OPT FNop("+",6,"ADD")
1Ø69Ø OPT FNop("-",6,"SUBTRACT")
1Ø7ØØ OPT FNop("=",4,"=")
1Ø71Ø OPT FNop("<",4,"<")
1Ø72Ø OPT FNop(">",4,">")
1Ø73Ø OPT FNop(CHR$&A4,4,"<>")
1Ø74Ø OPT FNop(CHR$&AØ,2,"AND")
1Ø75Ø OPT FNop(CHR$&A1,1,"OR")
1Ø76Ø OPT FNequb(255)
1Ø77Ø ]NEXT
1Ø78Ø ENDPROC
1Ø79Ø :
1Ø8ØØ DEF PROCsavecode
1Ø81Ø PROCoscli("SAVE TACT "+STR$~edstart%+"
      +"+STR$~edlen%+" "+STR$~tact)
1Ø82Ø PROCcentre(19,"Editor/Compiler saved")
1Ø83Ø ENDPROC
```

```
1Ø84Ø :
1Ø85Ø DEF PROCoscli($os%)
1Ø86Ø PROCcentre(17,"*"+$os%)
1Ø87Ø X%=os%:Y%=os% DIV 256
1Ø88Ø CALL oscli
1Ø89Ø ENDPROC
1Ø9ØØ :
1Ø91Ø DEF FNequb(byte%)
1Ø92Ø ?P%=byte%:P%=P%+1:=pass%
1Ø93Ø :
1Ø94Ø DEF FNequs(str$)
1Ø95Ø $P%=str$:P%=P%+LENstr$:=pass%
1Ø96Ø :
1Ø97Ø DEF FNequr(str$)
1Ø98Ø d%=FNequs(str$):P%=P%+1:=pass%
1Ø99Ø :
11ØØØ DEF FNequw(word%)
11Ø1Ø ?P%=word%:P%?1=word% DIV 256
11Ø2Ø P%=P%+2:=pass%
11Ø3Ø :
11Ø4Ø DEF PROCcentre(y,t$)
11Ø5Ø PRINTTAB(Ø,y)SPC4Ø;
11Ø6Ø PRINTTAB(2Ø-LENt$/2,y)t$;
11Ø7Ø ENDPROC
11Ø8Ø :
11Ø9Ø DEF FNtok(tok$,tok%,addr%)
111ØØ d%=FNequs(tok$)+FNequb(tok%)+FNequw(addr%)
1111Ø =pass%
1112Ø :
1113Ø DEF FNlib(key$)
1114Ø addr%=FNlibaddr(key$)
1115Ø [OPT pass%
1116Ø JSR addr%
1117Ø ]:=pass%
1118Ø :
1119Ø DEF FNasslib(key$)
112ØØ addr%=FNlibaddr(key$)
1121Ø [OPT pass%
1122Ø LDA #&2Ø
1123Ø JSR compbyte
1124Ø LDA #addr% MOD 256
1125Ø JSR compbyte
1126Ø LDA #addr% DIV 256
```

```
1127Ø JSR compbyte
1128Ø ]:=pass%
1129Ø :
113ØØ DEF FNlibaddr(key$)
1131Ø k%=-1:REPEATk%=k%+1
1132Ø UNTIL key$=lib$(k%) OR lib$(k%)="@"
1133Ø IF lib$(k%)="@" CLS:PROCcentre(2,"No such
     library routine as "+key$):END
1134Ø =lib%(k%)
1135Ø :
1136Ø DEF FNop(op$,pr%,oplib$)
1137Ø d%=FNequs(op$)+FNequb(pr%)+FNequw
     (FNlibaddr(oplib$))
1138Ø =pass%
1139Ø DEF FNerr(e$)
114ØØ d%=FNequw(Ø)+FNequs(e$)+FNequb(Ø)
1141Ø =pass%
1142Ø :
1143Ø DEF PROCreport
1144Ø CLS:PROCcentre(2,"Error")
1145Ø CLOSE#Ø
1146Ø PRINT'':REPORT
1147Ø PRINT" at line ";ERL
1148Ø END
```

Listing 16.2

```
 1Ø NOTE This is a demo
 2Ø NUMERIC count
 3Ø count=Ø
 4Ø LOOP
 5Ø count=count+1
 6Ø PRINT "Count=";
 7Ø PRINT count
 8Ø WHILE count<1ØØ
 9Ø VDU 7
1ØØ .next
11Ø IF count<>Ø
12Ø PRINT "Currently=";
13Ø PRINT count
14Ø count=count-1
```

```
150 JUMPTO next
160 CONTINUE
170 PRINT "Alldone"
180 END
```

Listing 16.3

```
 10 REM Display addresses
 20 REM Bruce Smith
 30 REM B/B+/E/M
 40 REM (C) Bruce Smith 1986
 50 :
 60 DIM lib$(40),lib%(40)
 70 in%=OPENUP"LIBINFO"
 80 INPUT#in%,stack%,stackend%
 90 INPUT#in%,liblen%
100 lowest%=stackend%+liblen%
110 k%=0
120 REPEAT
130 INPUT#in%,lib$(k%),lib%(k%)
140 k%=k%+1
150 UNTIL lib$(k%-1)="@"
160 :
170 CLOSE#in%
180 k%=0
190 REPEAT
200 PRINT lib$(k%);"=";~lib%(k%)
210 k%=k%+1
220 UNTIL lib$(k%-1)="@"
230 END
```

APPENDIX A

LONG ASSEMBLY LANGUAGE PROGRAMS

In writing the TACTLESS Compiler/Editor we have come almost to the very limits of what we can have in the way of having long assembly language programs in memory while still having room to assemble their code. Indeed, even with TACTLESS it has been necessary to have two other files already prepared and saved on disc before use.

We know we have used up all the memory available when either a 'no room' error is generated or when the assembler returns a error such as 'No such variable in line xx', when *we* know there is indeed such a variable. This happens because the extra hidden memory requirements for running a program, ie variable storage, have been gobbled up. So what can we do to get around this? There are two answers. The first is to obtain a special ROM called a Macro Assembler which allows you to write programs using a built-in editor, rather like a word processor, and then assemble them by reading each file in from disc. The assembled code itself may then also be written to disc. A second approach is to write a program which will do a similar job, and that is the approach outlined here.

The Linker

I have called the program which performs this macro assembly LINKER. It takes advantage of the fact that more than one program can be resident in the Micro's memory at once, simply by altering the value of PAGE. The Linker program controls the loading in of each sub-program at a different value of PAGE, calling this and taking control once again when the sub-program

has completed its task; the next sub-program can be loaded in and the process repeated.

Figure A.1 shows a typical memory arrangement for this system:

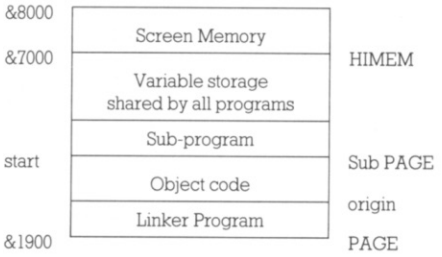


Figure A.1: The Macro Assembler Memory Map

Assuming PAGE is normally &1900, the Linker program is stored here; the variable *origin* is used to mark the start of the machine code assembly, ie the value P% is set to. Above this we have *start* or Sub-PAGE, the place in memory where each sub-program is loaded. All programs store their variables below HIMEM, and to make available as much space as possible the smallest screen mode should be selected. How much space you allocate to each section will of course depend on your needs. Ideally *origin* can be set at &3000 and a 4k block left to take the assembling machine code, which will put the value of *start* at &4000. How much space you leave for your sub-programs does of course depend on how long the longest one will be – in general, though, try to keep them to under 4k. Variables will reside from the TOP of this program to the bottom of the screen.

The Linker program is given in Listing A.1, and listings A.2, A.3, A.4 and A.5 are all short demo routines. Enter these as shown and save each one using the filename given in the REM statement at the start of each program.

PROCrun in the Linker program controls the movement of each sub-program into memory. The *files$* assignment in line 130 defines the order of the

source programs. As each source program is saved as SOURCEa, SOURCEB etc, each letter is in turn read from file$ and appended to the end of SOURCE in line 370 to load the correct program to *start*. Once here, PAGE is set to *start* and the sub-program is called as a subroutine with GOSUB 0. Thus it is vitally important that each sub-program begins with a line 0 and ends with a RETURN. The only other requirement is that each item of assembly code needs to have the OPT pass directive at its front end.

When the Linker program is RUN it will first of all read in the SOURCEI file – the initialisation file in which all variables to be used by the program should be stored. After this the main loop of the program is entered and this will read in each of the SOURCE files in turn – this effects the first pass of the assembler. Then they are all read in and run for a second time – thus completing the second pass of the assembler. On completion the start and end of the assembled machine code will be displayed.

This technique can still be used with tapes, the only requirement being that the files must be in the order needed on the tape. In the example here you should save the files on a blank tape in the following sequence:

SOURCEI, SOURCEA, SOURCEB, SOURCEC

When the first pass has been completed the tape will need to be re-wound to allow them to be read in for the second time. In such a case it is worth inserting an extra line into the linker program to prompt you to do this, thus:

245 PRINT TAB(4,6);"Rewind tape!":VDU 7:A=GET

Listing A.1

```
10 REM Macro Assembler - Linker
20 REM Bruce Smith
30 REM B/B+/E/M
40 REM (C) Bruce Smith 1986
50 :
60 MODE 6
70 DIM X% 255
80 PRINT SPC(10);"Macro Assembler"
```

```
 9Ø page%=PAGE
1ØØ origin=&3ØØØ
11Ø :
12Ø REM Place file letters here
13Ø file$="ABC"
14Ø :
15Ø REM Set variables first
16Ø PROCrun("I",&5ØØØ)
17Ø :
18Ø REM Now assemble source
19Ø PRINT TAB(4,6);"Pass 1"
2ØØ FOR pass=Ø TO 2 STEP 2
21Ø P%=origin
22Ø FOR files=1 TO LEN(file$)
23Ø PROCrun(MID$(file$,files,1),&5ØØØ)
24Ø NEXT files
25Ø PRINTTAB(4,6);"Pass 2"
26Ø NEXT pass
27Ø :
28Ø PRINT TAB(4,15)
29Ø PRINT "Object code at:
        ";STR$~origin;" TO ";STR$~P%
3ØØ PRINT "*SAVE code ";~origin;" ";~P%+1
31Ø PAGE=page%
32Ø END
33Ø :
34Ø DEF PROCrun (name$,start)
35Ø PRINT TAB(4,1Ø);
36Ø PRINT "Assembling: ";name$
37Ø PROCoscli("LOAD SOURCE"+name$+" "+STR$~start)
38Ø PAGE=start
39Ø GOSUB Ø
4ØØ ENDPROC
41Ø :
42Ø DEF PROCoscli(A$)
43Ø Y%=X% DIV 256
44Ø $X%=A$
45Ø CALL &FFF7
46Ø ENDPROC
```

Listing A.2

```
 Ø REM SET
 1Ø osasci=&FFE3
 2Ø oswrch=&FFEE
 3Ø osnewl=&FFE7
 4Ø osbyte=&FFF4
 5Ø prompt$="Hello there!"
 6Ø RETURN
```

Listing A.3

```
 Ø REM SOURCEA
 1Ø [OPT pass
 2Ø LDY #Ø
 3Ø .loop
 4Ø LDA text,Y
 5Ø JSR osasci
 6Ø INY
 7Ø CMP #13
 8Ø BNE loop
 9Ø ]
1ØØ RETURN
```

Listing A.4

```
 Ø REM SOURCEB
 1Ø [OPT pass
 2Ø LDA #7
 3Ø JSR oswrch
 4Ø JMP over
 5Ø \
 6Ø .text
 7Ø EQUS(prompt$)
 8Ø EQUB 13
 9Ø ]
1ØØ RETURN
```

Listing A.5

```
 Ø REM SOURCEC
 1Ø [OPT pass
 2Ø .over
 3Ø LDY #Ø
 4Ø .loop2
 5Ø LDA text2,Y
 6Ø JSR osasci
 7Ø INY
```

```
 8Ø CMP #13
 9Ø BNE loop2
1ØØ RTS
11Ø :
12Ø .text2
13Ø EQUS "Alldone!"
14Ø EQUB 13
15Ø ]
16Ø RETURN
```

STANDARD CONVERSION ROUTINES

Base conversion is a thorny task, so to help you along the way some standard routines are listed on the next couple of pages. The programs are:

Listing B.1: Single ASCII character to 8-bit binary
Listing B.2: Signed ASCII string to 16-bit binary
Listing B.3: Print 16-bit number as signed decimal string

Each listing contains a demonstration of use, and details of other conversion routines can be found in Appendix E.

Listing B.1

```
10 REM ASCII to 8 bit
20 REM Mastering Interpreters
30 REM and Compilers
40 REM (C) Bruce Smith 1986
50 :
60 PROCass
70 PRINT"Press any Hex key (ie 0-F) and"
80 PRINT"its hex value will be printed"
90 CALL test
100 PRINT" ";?&70;
110 END
120 :
130 DEF PROCass
140 FOR pass=0 TO 2 STEP 2
150 P%=&900
160 [OPT pass
170 .ascii_binary
180 CMP #48          \ is it >="0"
190 BCC illegal      \ control code
```

```
2ØØ CMP #58          \ is it Ø to 9?
21Ø BCC zero_nine    \ yes, skip A to F translation
22Ø SBC #7           \ jump to A to F codes
23Ø BCC illegal      \ carry set if legal
24Ø CMP #64          \ is it higher than '@'
25Ø BCS return       \ no, illegal so return
26Ø .zero_nine
27Ø AND #&ØF         \ preserve low nibble
28Ø .return
29Ø RTS
3ØØ .illegal
31Ø SEC              \ flag error
32Ø RTS
33Ø ] NEXT pass
34Ø :
35Ø REM Program Test Routine
36Ø FOR pass=Ø TO 2 STEP 2
37Ø P%=&98Ø
38Ø [OPT pass
39Ø .test
4ØØ JSR &FFEØ    \ osrdch
41Ø JSR ascii_binary
42Ø BCC over     \ no errors
43Ø LDA #&FF     \ Carry set - error
44Ø .over
45Ø STA &7Ø
46Ø RTS
47Ø ] NEXT pass
48Ø ENDPROC
```

Listing B.2

```
 1Ø REM ASCII decimal to 16 bit
 2Ø REM Mastering Interpreters
 3Ø REM and Compilers
 4Ø REM (C) Bruce Smith 1986
 5Ø :
 6Ø @%=Ø
 7Ø string_index=&7Ø
 8Ø current=&71
 9Ø sign_flag=&73
1ØØ buffer=&74
11Ø PROCass
12Ø PRINT" Enter number: ";
```

```
13Ø INPUT A$
14Ø length%=LEN(A$)
15Ø ?buffer=length%
16Ø FOR loop=1 TO length%+1
17Ø ascii=ASC(MID$(A$,loop,1))
18Ø buffer?loop=ascii
19Ø NEXT
2ØØ CALL decimal_string_hex
21Ø PRINT" Hex value is : &";
22Ø PRINT~(!current AND &FFFF)
23Ø END
24Ø :
25Ø DEF PROCass
26Ø FOR pass=Ø TO 2 STEP 2
27Ø P%=&9ØØ
28Ø [OPT pass
29Ø .decimal_string_hex
3ØØ LDX buffer
31Ø BEQ error
32Ø LDY #Ø
33Ø STY sign_flag
34Ø STY current
35Ø STY current+1
36Ø INY
37Ø STY string_index
38Ø LDA buffer,Y
39Ø CMP #ASC"-"
4ØØ BNE positive
41Ø LDA #&FF
42Ø STA sign_flag
43Ø INC string_index
44Ø DEX
45Ø BEQ error
46Ø JMP convert_character
47Ø :
48Ø .positive
49Ø CMP #ASC"+"
5ØØ BNE check_legality
51Ø INC string_index
52Ø DEX
53Ø BEQ error
54Ø .convert_character
55Ø LDY string_index
```

```
560 LDA buffer,Y
570 .check_legality
580 CMP #ASC"9"+1
590 BPL error
600 CMP #ASC"0"
610 BMI error
620 PHA
630 ASL current
640 ROL current+1
650 LDA current
660 LDY current+1
670 ASL current
680 ROL current+1
690 ASL current
700 ROL current+1
710 CLC
720 ADC current
730 STA current
740 TYA
750 ADC current+1
760 STA current+1
770 SEC
780 PLA
790 SBC #ASC"0"
800 CLC
810 ADC current
820 STA current
830 BCC no_carry
840 INC current+1
850 .no_carry
860 INC string_index
870 DEX
880 BNE convert_character
890 LDA sign_flag
900 BPL no_error
910 SEC
920 LDA #0
930 SBC current
940 STA current
950 LDA #0
960 SBC current+1
970 STA current+1
980 .no_error
```

```
 99Ø CLC
1ØØØ BCC finish
1Ø1Ø .error
1Ø2Ø SEC
1Ø3Ø .finish
1Ø4Ø RTS
1Ø5Ø ]
1Ø6Ø NEXT pass
1Ø7Ø ENDPROC
```

Listing B.3

```
 1Ø REM 16 bit to signed decimal
 2Ø REM Mastering Interpreters
 3Ø REM and Compilers
 4Ø REM (C) Bruce Smith 1986
 5Ø :
 6Ø binary_address=&7Ø
 7Ø temporary=&72
 8Ø sign_flag=&74
 9Ø buffer=&75
1ØØ PRINT"Type the hex value: ";
11Ø INPUT" &" hex$
12Ø hex$="&"+hex$
13Ø hex%=EVAL(hex$)
14Ø !binary_address=hex%
15Ø PROCass
16Ø CALL hex_decimal_string
17Ø PRINT "In decimal that is: ";
18Ø CALL string_print
19Ø END
2ØØ :
21Ø DEF PROCass
22Ø FOR pass=Ø TO 2 STEP 2
23Ø P%=&CØØ
24Ø [OPT pass
25Ø .hex_decimal_string
26Ø LDY #Ø
27Ø TYA
28Ø STA buffer
29Ø STA buffer+1
3ØØ STA buffer+2
```

```
310 STA buffer+3
320 STA buffer+4
330 LDA binary_address+1
340 STA sign_flag
350 BPL conversion
360 SEC
370 TYA
380 SBC binary_address
390 STA binary_address
400 TYA
410 SBC binary_address+1
420 STA binary_address+1
430 :
440 .conversion
450 LDA #0
460 STA temporary
470 STA temporary+1
480 CLC
490 LDX #16
500 .loop
510 ROL binary_address
520 ROL binary_address+1
530 ROL temporary
540 ROL temporary+1
550 SEC
560 LDA temporary
570 SBC #10
580 TAY
590 LDA temporary+1
600 SBC #0
610 BCC less_than
620 STY temporary
630 STA temporary+1
640 :
650 .less_than
660 DEX
670 BNE loop
680 ROL binary_address
690 ROL binary_address+1
700 :
710 .add_ascii
720 CLC
730 LDA temporary
```

```
74Ø ADC #ASC"Ø"
75Ø JSR concatenate
76Ø LDA binary_address
77Ø ORA binary_address+1
78Ø BNE conversion
79Ø :
8ØØ .finished
81Ø LDA sign_flag
82Ø BPL positive
83Ø LDA #ASC"-"
84Ø JSR concatenate
85Ø :
86Ø .positive
87Ø RTS
88Ø :
89Ø \ Form ASCII character string
9ØØ .concatenate
91Ø PHA
92Ø LDY#Ø
93Ø LDA buffer,Y
94Ø TAY
95Ø BEQ zero_finish
96Ø :
97Ø .shuffle_along
98Ø LDA buffer,Y
99Ø INY
1ØØØ STA buffer,Y
1Ø1Ø DEY : DEY
1Ø2Ø BNE shuffle_along
1Ø3Ø :
1Ø4Ø .zero_finish
1Ø5Ø PLA
1Ø6Ø LDY#1
1Ø7Ø STA buffer,Y
1Ø8Ø DEY
1Ø9Ø LDX buffer,Y
11ØØ INX
111Ø STX buffer,Y
112Ø RTS
113Ø :
114Ø \ string printing routine
115Ø .string_print
116Ø LDX buffer
```

```
117Ø LDY #1
118Ø .print_loop
119Ø LDA buffer,Y
12ØØ JSR &FFE3
121Ø INY : DEX
122Ø BNE print_loop
123Ø RTS
124Ø ]NEXT pass
125Ø ENDPROC
```

APPENDIX C

WHAT NEXT?

As mentioned in the introduction, *Mastering Interpreters and Compilers* is very much a practice rather than theory text. If you want to learn more about the theory of Interpreters and, in particular, Compilers, there are several books on the subject. I would recommend those detailed below as further reading.

If you are looking to expanding your interpreter or compiler into sideways RAM, there are at present just two books available on the subject, though modesty prevents me from making any comment on them!

All books should be readily available from any good computer bookshop or you can order them from their publisher quoting the ISBN number.

Understanding and Writing Compilers
by Richard Bornat
Published by Macmillan
ISBN 0-333-21732-2
Comments: Best of the Compiler books

Principle of Compiler Design
by A. V. Aho
Published by Addison-Wesley
Comments: Heavy going but a good read

Writing Interactive Compilers and Interpreters
by P. J. Brown
Published by Wiley
ISBN 0-471-27609-X
Comments: A good light read

Threaded Interpretive Languages
by R. G. Loeliger
Published by McGraw Hill
ISBN 0-07-038360-X
Comments: How a Forth Language works, based on the Z80, but useful all the same.

The BBC Micro ROM Book
by Bruce Smith
Published by Collins
ISBN 0-00-383075-6

The Advanced Sideways RAM Guide
by Bruce Smith
Published by Victory Publishing
ISBN 0-948938-00-5
(see Appendix E)

Software:

If you are interested in conversion routines to extend your own library, a disc of them is available from Lovebyte at 7.95 inclusive. For further details write to the author at the address given in Appendix E.

The FORTH Interest Group is highly recommended to anyone whose appetite has been whetted by the section on TILs. It produces a bi-monthly magazine called 'Forthwrite' and meetings are currently held once a month. In addition a FORTH source listing for the 6502 can be obtained from them along with friendly advice on getting it installed on your own machine!

For further details write to:

The Membership Secretary
Forth Interest Group
54 Wild Brier
Wokingham Berks R611 4UL

THE LIBRARY ROUTINES

The TACTLESS Library routines are used extensively by the compiled code. Extending the TACTLESS Compiler will mean use of the library routines and a list of these is given below to aid your coding.

To write code which will compile a JSR to a particular routine it is necessary to use the *FNasslib* function. For example, if you wanted to compile code that will subtract two numbers on the top of the data stack you would need to incorporate the following line into the source listing:

```
OPT FNasslib("SUBTRACT")
```

Chapter 15 contains details on how the existing commands were written; study it for further examples. Remember that all routines which work on numbers on the stack will leave a result on the stack.

The Library routines:

Name	Description
SETSTACK	Initialise stack
INPUTBUFF	Read a line from the keyboard
SKIP	Move r1 to first non-space character
CONV	Convert ASCII string to 16-bit number
NUM	Test to see if value is a number
PUSHDIR	Push next two bytes on data stack
PUSHIND	Push two bytes at following address

PULLIND	Pull two bytes and place at following addr
PUSH1	Push r1 on stack
PUSH2	Push r2 on stack
PUSH3	Push r3 on stack
ADD	Add 2 numbers on stack; place result on stack
SUBTRACT	As for ADD, but perform subtraction
MULT	As for ADD but perform a multiplication
PULL1	Pull stack and put into r1
PULL2	Pull stack and put into r2
PULL3	Pull stack and put into r3
INC1	Add 1 to r1
INC2	Add 1 to r2
INC3	Add 1 to r3
DEC1	Subtract 1 from r1
DEC2	Subtract 1 from r2
OSC	Pass line to *Oscli*
PRSTR	Print ASCII string that follows
DIV	Divide 1st number by 2nd and place result on stack
MOD	Leave remainder of DIV (as above) on stack
PRNUM	Print 16-bit value on stack as ASCII number
TRUE	Push &FFFF on stack
FALSE	Push &0000 on stack
FLAGTRUE	Set Z flag if number on stack is non-zero
<	Is 1st number on stack < 2nd number?
>	Is 1st number on stack > 2nd?
=	Are 2 numbers on stack equal?
<>	Are 2 numbers on stack not equal?
SCREEN	Select Mode using number on stack
VDU	Send number on stack to VDU stream
AND	Logically AND 2 numbers on stack
OR	Logically OR two numbers on stack

APPENDIX E

SOFTWARE

The Disc

A disc containing programs from this book is available for £7.95 including p&p, and on 3.5" disc for the Master Compact and Electron Plus 3; price £9.95.

The disc contains several extra items not in the book, including several TACTLESS demonstration programs and TACTLESS in the form of a ROM image which can be run using the ROM Filing System within sideways RAM.

To obtain a copy of the disc send a cheque or PO payable to 'Lovebyte' at:

Lovebyte
Victory Publishing
PO BOX 19
New Southgate
London
N11 1DS

INDEX

adding new dictionary entries 115
address – two-byte 23
ADFS (Advanced Disc Filing System) 53
analyse 244
anatomy 256
ASCII 16
  decimal string 213
ASL instruction 31
assembly language 11
  long programs 291

Bad mode 47
BASIC 14
  interpreter 10
BASIC Editor 178
BBC Monitor ROM 42
BBC Publications 42
BCPL 11, 163
BEEP 22, 26, 28
  TIL command 113
bit pattern 100
BMI instruction 23
body 106
BPL instruction 21, 23
BREAK key, defining of 53
Break, type of 87
BRK
  action on 83
  flag 83
  handling 81
  instruction 16, 81
BRKV 14, 82, 84, 113
  redirecting 85
building dictionary entries 143

CALL command 81
call interpreter listing 21
CALLSUB – TACTLESS command 238
CAT 16
cfa 107, 115, 117, 139
character 95
Charles Moore 135
checksums 7
CLEAR – TIL command 117
CLI instruction
CLIV 13, 14, 15, 28
compiler registers 172
code field address 107
code field pointer 116
COLON – FORTH routine 140
command 13, 14, 15, 28, 44, 60
  execution address 106
  execution address tables 30
  identification 30
  identify 20
  jump to execution address 20
  line interpreter vector 13
  move to next 20
  not known 20
  not recognised 21
  adding new ones 97
  immediate 49
  syntax 49
  table 19
  table, end of 22
  table index 21
  table makeup 22
  to set GTOP 93
comments 43
compatibility 7
compilation 164
compiler 262
  analyse 244
  an introduction 163
  conditional 170
  data stack 171
  direct threaded code 174
  TACTLESS direct commands 225
  TACTLESS library indirect routines 181
  information block 175
  jumps 170
  lexical analysis 165
  library 166
  library 173
  library 178
  loop control 170
  multi-pass 165
  single-pass 165
  source language 164, 170
  source language line numbers 171
  subroutines 170
  symbol table 165
  syntax analysis 165
TACTLESS 168, 244
TACTLESS anatomy 256
TACTLESS commands 207, 250
TACTLESS data stack 181
TACTLESS detokenising 209
TACTLESS editor 200
TACTLESS Electron notes 254
TACTLESS errors 186, 251
TACTLESS extending 262
TACTLESS first pass 245
TACTLESS handling numbers 213
TACTLESS LIBCODE 184
TACTLESS LIBINFO file 179, 200, 214
TACTLESS library indirect routines 187
TACTLESS library routines 307–308
TACTLESS library routines precis 184
TACTLESS not direct error 253
TACTLESS operators 207, 243
TACTLESS program commands 229
TACTLESS RPN handler 213, 233
TACTLESS safety net 249
TACTLESS second pass 249
TACTLESS tokeniser 206
TACTLESS using 252
TACTLESS variables 218
TACTLESS variable space allocation 219
TACTLESS language 224

translate 244
translation 165
two-pass 165
variables 170
variable type bytes 175
writing 168
computation stack 136
CONTINUE TACTLESS command 241
conversion – hex-to-binary 51
conversion routines 297
CREATE FORTH word 143
CSL 170

data stack 136, 171–172, 181
debugging 81
delete 87
detokenising 211
dictionary-entries, adding new ones 115
direct commands 170
direct library routines 181
direct threaded code 174
disc of programs 7, 309
dictionary entry 106
DOCOL FORTH routine 139
DUP FORTH command 137, 143
dynamic variables 49

Econet workspace 46
editor 200
end of command table 22
END TACTLESS command 237, 249
entry conditions 14
EOR 208
EOR rules 112
EQU simulations 8
error message 16, 186
GRAFRITE format 83
Bad mode 47
Command not known 52
ESCAPE 59, 85, 87, 96
ESCAPE key – detection 58
EXECUTE FORTH primitive 141
execution address 20, 22
exit conditions 14

FIG FORTH user group 306
figure 2.1 CLI details 14
figure 3.1 20
figure 3.2 command table storage 23
figure 3.3 bit patterns 23
figure 3.4 the new command table 24
figure 4.1 pseudo flowchart 43–44
figure 4.2 free RAM areas 45–46
figure 4.3 GRAFRITE memory map 46
figure 7.1 a typical TIL entry 107
figure 7.2 how dictionary entries are linked 108
figure 7.3 113
figure 7.4 flow of a FORTH word 118
figure 8.1 inner interpreter routines 139
figure 8.2 comparison of secondary and primary 141
figure 8.3 typial FORTH memory map 147
figure 10.1 – the classic compiler 166
figure 11.1 – memory map part 1 172
figure 11.2 memory map part 2 173
figure 11.3 memory map part 3 173
figure 11.4 memory map part 4 174
figure 14.1 stucture of variable entry 221
figure 16.1 an idealised two-pass compiler 244
figure 16.2 disassembly of listing 16.2 251–258
figure A.1 the macro assembler 292
flowchart 42
FORTH 11, 115, 116, 135, 164
constants 144
dictionary format 138
variables 144
function key buffer 46
further reading 305–306

GRAFRITE
adding new commands 97
writing 52
improvements 93
initialisation 54
memory map 46
object code 53
program loop 50
program editing 95
prompt 56
source program 42
the language 41
control loop 50
graphics language 41
GTOP 47, 56, 59, 93
guidelines for writing a language 41

handling BRK 81
handling numbers 213
hard BREAK 87
header 106
help 16, 26, 28
hexadecimal values 51
hex-to-binary conversion 51
HIMEM 45, 47, 173
HOME 27
HOME TIL command 115

identified command 17, 20
INFO TACTLESS command 241
immediate command 49, 60
index registers 14
indirect index 20
indirect library routines 181
information block 175
IF TACTLESS command 228
inner interpreter 138
INPUT TACTLESS command 239
inserting the wedge 28
interpretive pointer 116
interpretation 10
interpreter 10
interpreter core 45
interpretive call 25
interrupt 17
InterWord 95
IP 116
IRQ command 83

JUMPTO TACTLESS command 239

keyword 106
keywords 17

languages – effect of BREAK 81
language TACTLESS 224
language workspace 45
lexical analysis 165
LFA 107, 110
LIBCODE 184
LIBINFO file 179, 200
library construction 166

library routines 184, 307–308
LIFO 136
line numbers 171
linear construction 166
link address 106
linker program 291
link field address 107
LISP 11
LIST GRAFRITE command 52
listing 3.1 31
listing 3.2 29, 36
listing 4.1 GRAFRITE interpreter 62
listing 5.2 error messages 82
listing 5.3 intercepting BRK 88
listing 6.1 VIEW/Wordwise formatter 103
listing 7.1 miniTIL 121
listing 7.2 miniTHORF 127
listing 8.1 data stack simulation 147
listing 9.1 the RPN stack 156
listing B.1 ASCII to binary 297
listing B.2 signed ASCII to 16-bit binary 298
listing B.3 print 16-bit number 301
listing 11.1 macro demonstration 177
listing 12.1 the TACTLESS library 183
listing 16.1 TACTLESS Editor 262
listing 16.2 a demo TACTLESS program 289
listing 16.3 display addresses 290
listings disc 309
LIST TACTLESS command 225
LOAD 52, 93
LOAD TACTLESS command 229
logically ANDing 24
long programs 291
LOOP TACTLESS command 235
LOUD 22, 26
Lukasiewicz 152

machine code 11
machine code monitor 42
Machine Operating System 13
macro 214
macros 176
main compiled code (MCC) 171
marker byte 20
marker character in GRAFRITE 102
MCC 171
memory map 12, 45
memory pointers 43
Mistake error 81
MOS 13
mount 53
multi-pass compiler 165

name field address (NFA) 107
negative byte 24, 55, 99
NEW TACTLESS command 225
NEXT FORTH routine 139
NFA 107, 110
NOP instruction 54
NORM 26
notebook 41
NOTE TACTLESS command 230
NUMERIC TACTLESS command 233

offset 31
operating system vector 12
operation interception 12
OSASCI 16
OSBYTE 58, 87
OSBYTE call 27, 172
OSCLI 14
OSHWM 45, 47, 48, 146, 172, 180
OSWORD 0 110, 200
OSWORD 42, 57, 59
OSWRCH 42
outer interpreter 138

PAGE 52, 53
PASCAL 11, 163
phases 164
PHP instruction 15
planning 43
PLP instruction 16
power-up reset 87
precedence table 216
primitives 119
PRINT TACTLESS command 230
revectoring CLI 15
program errors 52
program editing 95
programs disc 310
pseudo code 43, 162

RAM 14
read line to memory 202
redirecting BRKV 85
REPEAT GRAFRITE command 50
repeating a loop 101
return stack 136
revectoring 14, 16
Reverse Polish Notation (RPN) 151
ROM filing system 256
RPN handler 213, 233
RPN routine 215
RTI instruction – use of 14

SAVE 29, 52, 93
SAVE TACTLESS command 229
SCREEN GRAFRITE command 50, 52, 235
searching dictionary 110
secondaries 119
SEI instruction 15
SEMI FORTH primitive 140
SET GRAFRITE command 94
single pass compiler 165
SIXMODE TIL command 115
SOFT 22, 26
soft BREAK 87
software disc 310
source program GRAFRITE 52
stack 83, 136
stack, control of 81
stack, loop in GRAFRITE 101
stack pointer 136
standard conversion routines 297
static variables 49
storage 47
string printing routine 54
structure of variables 48
sub phases 164
symbol table 165
syntax analysis 165
syntax checker 100

TAB 87
table 2.1 the OS vectors 13
table 4.1 provisional commands 44, 45
table 4.2 OSWORDO parameter block 57
table 4.3 contents of parameter block 58
table 5.1 routines described in text 87–88
table 5.2 routines not in text 88
table 6.1 BBC BASIC's tokens 100

table 7.1 the main routines of listing 7.1 114–115
table 7.2 the main routines in miniTHORF 120
table 8.1 the main routines of listing 8.1 143
table 9.1 standard versus RPN notation 152–153
table 9.2 the new routines in listing 9.1 155–156
table 15.1 TACTLESS language commands 224
TACTLESS – see under 'compiler'
TACT 253
tape users 225
the interpreter 19
THORF 119
threaded interpretive languages 106
TIL 106
  body 106
  building dictionary entries 143
  code field address 107, 116, 117
  code field pointer 116
  COLON FORTH routine 140
  command execution address 106
  CREATE FORTH word 143
  dictionary entry 106
  DOCOL FORTH routine 139
  DUP FORTH word 143
  EXECUTE FORTH primitive 141
  FORTH computation stack 136
  FORTH dictionary format 138
  FORTH return stack 136
  FORTH variables and constants 144
  header 106
  inner interpreter in FORTH 138
  interpretive pointer 116
  link address 106, 107, 110
  name field address 107, 110
  NEXT FORTH routine 139
  outer interpreter in Forth 138
  primitives 119
  searching dictionary 110
  secondaries 119
  SEMI FORTH primitive 140
  true one 115
tokenising 99, 206
tokenising process 208
tokens 29
tokens in GRAFRITE 99
tools to write with 42
TOP 47
top-down approach 52
translate 244
translation 165
true TIL 115
two-pass compiler 165
type of BREAK 87

unknown command 26
unknown error 81

Variables 48, 170
  operators 50, 51
  BBC BASIC 49
  distinguishing 48
  double-byte 48
  five-byte 48
  hash 48
  integer 48
  single-byte 48
  static 49
  syntax 50
  structure 48
VDU codes 42
VDU commands 97
VDU TACTLESS command 234
vector 12, 47
  BRK 14
  command line interpreter 13
  addresses 172
  pointer 20
  redirecting BRKV 85
  BRKV 82, 113
VIEW 95

W 116
wedge 16
WEDGE2 28
wedge inserting 28
wedge interpreter 12
WHILE GRAFRITE command 50, 52
WHILE TACTLESS command 235
word processor 95, 171
Wordwise 95

XSAVE 136

&F7 85
&F8 85
&FD 83, 84
&FF 21, 23, 56, 60, 62, 96, 171, 249
&FFF7 14
&FFFE 83
&COO 57

This clear and comprehensive introduction to compilers and interpreters emphasises the practical side of the art. It moves gradually from the idea of a 'wedge' in the BBC Computer's operating system, to a simple interpreter, a simple graphics language, threaded interpretive languages (including FORTH) and, finally, a stand-alone compiler. Listings of the implementations are given. This book will give anyone with a good knowledge of assembly language the foundation on which to build an interpreter or compiler of their own.

Bruce Smith has written sixteen books on computing, fourteen of which have dealt with the BBC Computer range. He is currently technical editor of *Acorn User* magazine.

£14·95

ISBN 0-563-21283-7